

1 2 9 0



UNIVERSIDADE D
COIMBRA

Paulo Jorge Costa Nunes

BLENDDED SECURITY ANALYSIS FOR WEB APPLICATIONS: TECHNIQUES AND TOOLS

Tese no âmbito do Programa de Doutoramento em Ciências
e Tecnologias da Informação orientada pelo
Professor Doutor José Carlos Martins da Fonseca e
Professor Doutor Marco Paulo Amorim Vieira e apresentada ao
Departamento de Engenharia Informática da Faculdade de
Ciências e Tecnologia da Universidade de Coimbra

Dezembro de 2021

With the advent of the Internet and Information Society, the popularity of web applications is increasing, resulting in rapid information growth and a clear impact on security aspects. In fact, web applications are frequently deployed with critical security vulnerabilities that, when exploited, may have a huge negative effect on the business. At the same time, headlines regularly chronicle technology based issues such as security hacks, inappropriate or illegal surveillance, misuse of personal data and spread of misinformation. The distrust these incidents breed in stakeholders, whether customers, employees, partners, investors, or regulators, can significantly damage the reputation of an organization.

This thesis addresses the problem of vulnerability detection in web applications, proposing new techniques and tools to significantly increase the detection rate of vulnerabilities while keeping the number of false positives low. For demonstration purposes, the work focuses on WordPress plugins and on the PHP language. WordPress is developed in PHP and it is the most popular Content Management System ([CMS](#)) nowadays, being used by 43.1% of all the websites (as of December 19th, 2021), which corresponds to a [CMS](#) market share of 65.2%. The WordPress environment has more vulnerabilities than any other [CMS](#). In fact, according to data from WPScan, approximately 97% of vulnerabilities in their database are related to WordPress plugins and themes and only 3% are core software.

Our first contribution is a Static Application Security Testing ([SAST](#)) tool, named `phpSAFE`, able to analyze PHP code in complex web applications that use Object-Oriented Programming ([OOP](#)) and are based on third-party plugins. `phpSAFE` is a follow-up of a project whose development was requested by Automattic, the developer of WordPress, with the goal of improving the security of the plugins. Unlike other tools, `phpSAFE` is deployed with a default configuration focusing on generic SQL Injection ([SQLi](#)) and Cross-Site Scripting ([XSS](#)) vulnerabilities, as well as on plugins of the WordPress framework. This solution, out-of-the-box, has the advantage of allowing the immediate use of the tool to analyze PHP code, either from applications or WordPress plugins without requiring further configuration. It may be used both by occasional developers and by professional software houses wanting to speed up the development process of more secure software

and reducing costs by avoiding the use expensive commercial [SAST](#) tools.

The second key contribution is a new benchmarking methodology for [SAST](#) tools for web security that considers four scenarios of criticality and uses evaluation metrics tuned for each scenario. This methodology can be used to compare and rank different [SAST](#) tools for detecting vulnerabilities in web applications according to different vulnerability detection requirements. In this context, a benchmark consists of four components: scenarios, workload, metrics, and procedures and rules. In practice, we consider four scenarios (highest-quality, high-quality, medium-quality and low-quality) to compose the workload by assigning representative applications to each scenario based on *code quality*, and rely on a *main metric* and a *tiebreaker metric* for characterizing the tools under evaluation in each scenario. The main metric is used to rank the [SAST](#) tools and the tiebreaker metric is used to decide eventual ties between two or more [SAST](#) tools. We created two concrete instances of the general benchmarking methodology to demonstrate its feasibility, and evaluated and compared five [SAST](#) tools for the detection of [SQLi](#) and [XSS](#) vulnerabilities in a workload composed of 134 WordPress plugins. Results show that, our benchmarking approach is a valuable tool to help project managers choosing the best [SAST](#) tool for a specific project, according to their needs and the resources available.

Another contribution consists on case studies on the combination of the results of five [SAST](#) tools for [SQLi](#) and [XSS](#) vulnerabilities, as a way to improve the vulnerability detection rate while keeping False Positives ([FPs](#)) low. First, we conducted an experimental campaign using *1-out-of-n* adjudication on the outputs of the [SAST](#) tools on a dataset of real WordPress plugins. The main limitation observed is the potential increase of [FPs](#), which may be unacceptable in many situations. Thus, we conducted an empirical study looking at the results of all the possible *1-out-of-n*, *n-out-of-n* and *majority voting* configurations. This way, we collect more evidence on the interplay between [FPs](#) and False Negatives ([FNs](#)) in diverse [SAST](#) configurations, being able to rank the best combination of tools. Finally, we performed an in-deep analysis of code of the WordPress plugins to find reasons to justify why some [SAST](#) tools do not detect vulnerabilities that other [SAST](#) tools detect.

The thesis ends with the proposal of a generic methodology for blending static and dynamic analysis for web application vulnerability detection. The methodology combines static analysis, a crawling procedure and dynamic analysis into a number of steps in order to obtain a set of vulnerabilities, reported by the static analysis, that are confirmed to be exploitable. In short, the process starts with Static Analysis ([SA](#)) to produce a list of candidate vulnerabilities. Then, the application is executed automatically, stopping only when the code where the vulnerabilities are located is run. A set of specific inputs and configuration options are automatically generated from the results of the [SA](#) and the runtime information collected, which are used to guide the Dynamic Analysis ([DA](#)) in the process of successfully exploiting each vulnerability reported by the [SA](#). Results show that, our approach is a great improvement for security practitioners, over using only [SA](#) and manual review, because it reduces the usual need for manual reviews of the output of [SAST](#) tools.

Keywords: benchmarking, dynamic analysis, security, [SQLi](#), [SQLi](#) attacks, static analysis, taint analysis, vulnerabilities, web applications, [XSS](#).

Com o advento da Internet e da Sociedade da Informação, a popularidade das aplicações web está aumentando, resultando num rápido crescimento da informação e um claro impacto nos aspectos de segurança. Na verdade, as aplicações web são frequentemente desenvolvidas com vulnerabilidades de segurança críticas que, quando exploradas, podem ter um grande efeito negativo nos negócios. Ao mesmo tempo, as manchetes relatam regularmente questões de base tecnológica, como hacks de segurança, vigilância inadequada ou ilegal, uso indevido de dados pessoais e disseminação de informações incorretas. A desconfiança que esses incidentes geram nas partes interessadas, sejam eles clientes, funcionários, parceiros, investidores ou reguladores, pode prejudicar significativamente a reputação de uma organização.

Esta tese aborda o problema de detecção de vulnerabilidades em aplicações web, propondo novas técnicas e ferramentas para aumentar significativamente a taxa de detecção de vulnerabilidades, mantendo baixo o número de falsos positivos (FPs). Para fins de demonstração, o trabalho é focado em plugins do WordPress e na linguagem PHP. O WordPress é desenvolvido em PHP e é o CMS mais popular da atualidade, sendo utilizado por 43,1% de todos os sites (19 de dezembro de 2021), o que corresponde a quota de mercado de 65,2% dos CMS. O framework WordPress tem mais vulnerabilidades do que qualquer outro CMS. Na verdade, de acordo com dados do WPScan, aproximadamente 97% das vulnerabilidades na sua base de dados estão relacionadas com plugins e temas do WordPress e apenas 3% são do core do WordPress.

A nossa primeira contribuição é uma ferramenta de análise estática de código fonte (SAST) com foco em segurança de aplicações web, chamada phpSAFE, capaz de analisar PHP código fonte de aplicações web complexas que usam OOP e baseadas em plugins de terceiros. phpSAFE é um follow-up de um projeto cujo desenvolvimento foi solicitado pela Automattic, desenvolvedora do WordPress, com o objetivo de melhorar a segurança dos plugins para WordPress. Ao contrário de outras ferramentas, phpSAFE é implantado com uma configuração padrão com foco em vulnerabilidades genéricas de SQL Injection (SQLi) e Cross-Site Scripting (XSS), bem como em plugins para o WordPress. Esta solução, out-of-the-box, tem a vantagem de permitir o uso imediato da ferramenta para analisar o código PHP de aplicações web ou plugins do

WordPress sem a necessidade de outras configurações. A Ferramenta pode ser usado quer por programadores ocasionais quer por software houses profissionais que desejam acelerar o processo de desenvolvimento de software mais seguro e reduzir custos, evitando o uso de ferramentas SAST comerciais caras. A segunda contribuição principal é uma nova metodologia de benchmarking para ferramentas SAST para segurança na web que considera quatro cenários de criticidade e usa métricas de avaliação ajustadas para cada cenário. Esta metodologia pode ser usada para comparar e classificar diferentes ferramentas SAST para detectar vulnerabilidades aplicações web de acordo com os diferentes requisitos de detecção de vulnerabilidades. Nesse contexto, um benchmark consiste em quatro componentes: cenários, carga de trabalho, métricas e procedimentos e regras. Na prática, consideramos quatro cenários (qualidade superior, alta qualidade, qualidade média e baixa qualidade) para compor a carga de trabalho atribuindo aplicações web representativas para cada cenário com base na qualidade do código, e é utilizada uma métrica principal e uma métrica de desempate para classificar as ferramentas em avaliação em cada cenário. A métrica principal é usada para classificar as ferramentas SAST e a métrica de desempate é usada apenas para decidir eventuais empates entre duas ou mais ferramentas SAST. Criámos duas instâncias concretas da metodologia geral de benchmarking para demonstrar sua viabilidade e avaliamos e comparamos cinco ferramentas SAST para a detecção de vulnerabilidades SQLi e XSS com uma carga de trabalho composta por 134 plugins do WordPress. Os resultados mostram que nossa abordagem de benchmarking é uma ferramenta valiosa para ajudar os gerentes de projeto a escolher a melhor ferramenta SAST para um projeto específico, de acordo com suas necessidades e os recursos disponíveis.

Outra contribuição consiste em estudos de caso sobre a combinação dos resultados de cinco ferramentas SAST para vulnerabilidades SQLi e XSS, como forma de melhorar a taxa de detecção de vulnerabilidade mantendo o número de FPs baixo. Primeiro, conduzimos uma campanha experimental usando uma adjudicação 1-out-of-n dos resultados das ferramentas SAST num conjunto de dados de plugins reais do WordPress. A principal limitação observada é o potencial aumento de FPs, que pode ser inaceitável em muitas situações. Assim, conduzimos um estudo empírico olhando os resultados de todas as adjudicações possíveis 1-de-n, n-de-n e votação por maioria. Desta forma, recolhemos mais evidências sobre a balanço entre FPs e falsos negativos (FNs) em diversas adjudicações de ferramentas SAST, podendo classificar a melhor combinação de ferramentas. Finalmente, realizamos uma análise profunda do código dos plugins do WordPress para encontrar razões para justificar porque algumas ferramentas SAST não detectam vulnerabilidades que outras ferramentas SAST detectam.

A tese termina com a proposta de uma metodologia genérica para combinar análise estática (SA) e análise dinâmica (DA) para detecção de vulnerabilidades em aplicações web. A metodologia combina a análise estática, um procedimento de crawling e análise dinâmica numa série de etapas a fim de obter um conjunto de vulnerabilidades, relatadas pela análise estática, que são confirmadas pela análise dinâmica como exploráveis. Resumindo, o processo começa com SA para produzir uma lista de vulnerabilidades candidatas. Em seguida, a aplicação web é executado automaticamente, parando apenas quando o código onde as vulnerabilidades estão localizadas for executado. Um conjunto de dados e opções de configuração específicas é gerado automaticamente a partir dos resultados da SA e das informações recolhidas durante execução da aplicação web, que são usadas para guiar a DA no processo de exploração bem-sucedida de cada vulnerabilidade relatada pelo SA. Os resultados mostram que a nossa abordagem é uma grande melhoria para

os profissionais de segurança, ao usar apenas SA e revisão manual, porque reduz a necessidade usual de revisões manuais da saída das ferramentas SAST.

Palavras Chave: análise de contaminação, análise dinâmica, análise estática, aplicações web, ataques de SQL Injection, segurança, SQL Injection, testes padronizados, XSS, vulnerabilidades.

Acknowledgement

Thanks to my advisors Professor José Fonseca and Professor Marco Vieira who enabled me to research on such a hot topic. I want to thank especially for their expertise, ideas, feedback, time, example of quality and dedication to research and the way they taught me to do serious and rigorous research work.

I also would like to thank my colleagues of the Department of Informatics Engineering of the Escola Superior de Tecnologia e Gestão of the Instituto Politécnico da Guarda for the excellent working environment.

I would also like to thank the Software and Systems Engineering Group of the Centre of Informatics and Systems of the University of Coimbra (SSE/CISUC) for their financial support, which helped me perform my research work. A word of appreciation to the members of the SSE/CISUC for the excellent quality of the work they develop.

I also thank to all the anonymous reviewers of the papers for their critical feedback and constructive comments that helped to improve the quality of the work done so far.

To my friends, I thank for their support.

I want to express my gratitude to my parents António and Maria who helped and encouraged me throughout my life.

I am very grateful to all my family for their encouragement and support during this long path.

Special tanks to all the people who along the way believed in me.

Last but not the least, I want to thank to my son Miguel and my love Nélia for their encouragement, support and understanding.

Guarda, December 2021.

Table of Contents	xii
1. Introduction	1
1.1. Context and Motivation	2
1.2. Thesis Contributions	6
1.3. Structure of the Thesis	7
2. Background and Related Work	9
2.1. Computer Security Concepts	9
2.2. Web Applications and Security	14
2.3. Common Web Application Vulnerabilities	17
2.3.1. Cross-Site Scripting	18
2.3.2. SQL Injection	25
2.4. Static Code Analysis for Vulnerability Detection	30
2.4.1. Control-flow and Data-flow Graphs	31
2.4.2. Static Analysis Techniques	34
2.4.3. Static Taint Analysis	38
2.4.4. Combining Static Analysis Tools	40
2.5. Dynamic and Hybrid Security Analysis	41
2.5.1. Taint-based Protection	42
2.5.2. Tainted-free Protection	43
2.5.3. Black-box and White-box Testing	44
2.5.4. Hybrid Analysis	45
2.6. Benchmarking	49
2.7. Conclusion	54
3. A Security Analysis Tool for OOP Web Application Plugins	56
3.1. Detection Approach and the phpSAFE Tool	57
3.1.1. Configuration Stage	58

3.1.2.	Model Construction Stage	59
3.1.3.	Analysis Stage	59
3.1.4.	Results Processing Stage	60
3.2.	Evaluation of phpSAFE	62
3.3.	Results and Discussion	64
3.3.1.	Overall Analysis	64
3.3.2.	Vulnerability Detection Overlap	65
3.3.3.	Inertia in Fixing Vulnerabilities	67
3.4.	Conclusion	68
4.	Benchmarking Static Analysis Tools for Web Security	70
4.1.	Benchmarking Approach	71
4.1.1.	Application Scenarios	72
4.1.2.	Benchmark Metrics	73
4.1.3.	Building the Workload	76
4.1.4.	Procedure and Rules	81
4.2.	Benchmark Instantiation	83
4.2.1.	Collecting the Source Code of Vulnerable Applications	83
4.2.2.	Assigning Applications to Scenarios	84
4.2.3.	Identifying Vulnerabilities and Non-vulnerabilities	85
4.3.	Experimental Evaluation	85
4.3.1.	Ranking the SAST Tools	86
4.3.2.	Results for SAMATE and BSA Metrics	89
4.3.3.	Limitations and Benchmark Properties	94
4.4.	Conclusion	96
5.	Combining Diverse SAST Tools for Web Security	98
5.1.	Case Study: 1-out-of-n Adjudication	99
5.1.1.	Hypotheses and Analysis Approach	100
5.1.2.	Results for SQLi Vulnerabilities	101
5.1.3.	Results for XSS Vulnerabilities	103
5.1.4.	Testing the Hypotheses	104
5.2.	Case Study: Diverse Adjudication Strategies	105
5.2.1.	Hypotheses and Analysis Approach	106
5.2.2.	Diversity of the Individual SAST Tools	108
5.2.3.	Results for Diverse SAST tools	113
5.2.4.	Testing the Hypotheses	117
5.2.5.	Identifying Strengths and Weaknesses of SAST Tools	118
5.3.	Threats to Validity	132
5.4.	Conclusion	133
6.	Blending Static and Dynamic Analysis for Vulnerability Detection	135
6.1.	Approach for Blending Static and Dynamic Analysis	136
6.1.1.	Obtaining Static Analysis Data	138
6.1.2.	Gathering Runtime Information	139

6.1.3.	Mapping HTTP Requests with Trace Files	140
6.1.4.	Generating the DA Configuration	140
6.1.5.	Testing Vulnerability Exploitability	142
6.1.6.	PoC Reporting	142
6.2.	Instantiation and Experimental Setup	142
6.2.1.	Obtaining Static Analysis Data	144
6.2.2.	Gathering Runtime Information	144
6.2.3.	Mapping HTTP Requests with Trace Files	146
6.2.4.	Generating the DA Configuration	147
6.2.5.	Testing the Vulnerability Exploitability	149
6.2.6.	PoC Reporting	150
6.3.	Results and Discussion	150
6.3.1.	Overall Results	151
6.3.2.	Testing the Vulnerability Exploitability	151
6.3.3.	Testing the Non-Exploitability of FPs	154
6.3.4.	Comparison with Alternative Approaches	155
6.3.5.	Threats to Validity	157
6.4.	Conclusion	157
7.	Conclusions and Future Work	159
7.1.	Key Contributions	160
7.2.	Future Work	163
A.	Assigning Applications to Scenarios	165
A.1.	Characterizing Software Quality	165
A.2.	Process for Assigning Applications to Scenarios	166
A.2.1.	The Quality Model	167
A.2.2.	Gathering the Source Code Metrics	171
A.2.3.	Deriving Ratings of Applications	174
A.3.	Rating Thresholds Tables	177
B.	List of WordPress plugins	179
C.	Benchmarking Procedure and Rules	183
C.1.	Preparation	183
C.2.	Execution	185
C.3.	Normalization of Reports	185
C.4.	Vulnerability Verification	185
C.5.	Metrics Calculation and Ranking	186
D.	Results for all Combinations of five SAST Tools: WordPress Plugins	188
E.	Case Study: Synthetic Dataset Using the 1-out-of-n Strategy	194
E.1.	Workload	195
E.1.1.	Collecting the Source Code of Vulnerable Applications	195
E.1.2.	Assigning Test Cases to Scenarios	198

E.1.3. Characterizing VLOCs and NVLOCs of Synthetic Test Cases	198
E.2. Benchmark Run	199
E.3. Results and Discussion	200
E.3.1. Comparing the Results of the WordPress Plugins Dataset and the Synthetic Dataset	200
E.3.2. Testing the Hypotheses	202
E.4. Conclusion	204
E.5. Best Solutions for the Synthetic Dataset	204
List of Abbreviations and Symbols	209

Publications

In the context of this doctoral research work, the following articles have been published in international journals with peer-reviewing:

- J1) **Paulo Nunes**, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia and Marco Vieira. “An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios”. In: *Computing* 101, 161-185 (Sept. 2018). ISSN: 1436-5057. DOI:10.1007/s00607-018-0664-z. <https://doi.org/10.1007/s00607-018-0664-z>.
- J2) **Paulo Nunes**, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia and Marco Vieira. “Benchmarking Static Analysis Tools for Web Security”. In: *IEEE Transactions on Reliability* vol. 67, no. 3, pp. 1159–1175 (Sept. 2018). ISSN: 0018-9529. DOI:10.1109/TR.2018.2839339. <https://doi.org/10.1109/TR.2018.2839339>

Furthermore, the following articles have been published in international conferences with peer-reviewing:

- C1) **Paulo Nunes**, José Fonseca, and Marco Vieira. “phpSAFE: A Security Analysis Tool for OOP Web Application Plugins”. In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Rio de Janeiro, June 2015, pp. 299-306. DOI: 10.1109/DSN.2015.16. <https://doi.org/10.1109/DSN.2015.16>
- C2) **Paulo Nunes**, Ibéria Medeiros, José. Fonseca, Nuno Neves, Miguel Correia and Marco Vieira, “On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study”. In *2017 13th European Dependable Computing Conference (EDCC)*, Geneva, 2017, pp. 121-128. DOI: 10.1109/EDCC.2017.16. <https://doi.org/10.1109/EDCC.2017.16>
- C3) Areej Algaith, **Paulo Nunes**, José Fonseca, Ilir Gashi and Marco Vieira, “Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools”. In *2018 14th European Dependable Computing Conference (EDCC)*, Iasi, Romania, 2018, pp. 57-64. DOI: 10.1109/EDCC.2018.00020. <https://doi.org/10.1109/EDCC.2018.00020>

Ranging from individuals to large organizations, almost everything is stored, available, discussed, shared, processed or traded on the web. In fact, the popularity of web applications is growing exponentially, making them unavoidable for most of our daily activities. Nowadays, millions of people use web applications as part of their lives by accessing and managing confidential information, including financial, health and personal data that must be reliable and readily accessible. A **web application** is a software program that is stored on a remote server and the result of the execution is delivered over the *Internet* through a web browser interface. Browsers come installed by default on almost any device with Internet connectivity, such as computers, tablets and cellphones, which contributes for the widely adoption of web applications, making them ubiquitous and used by almost everyone around the globe.

Modern web applications have a rich interface and a user experience that resembles the desktop counterpart. Due to the high demand, they are being created at an increasingly fast pace, with the help of easy to use and customizable frameworks, even by non-experts. Examples of common web applications include blogs, personal pages, games, webmail, online banking, e-commerce, healthcare, insurance, education, social networking, wikis, government and corporate sites, among many others.

Web applications became the most prevalent platform used by organizations of any size to provide business information and critical services to their clients and partners. Therefore, the need for complex web applications ready to deal with the intricacy, criticality and tight constraints of current business needs is tremendous. This pressures the development of web applications that are often delivered with insufficient security features, ignoring most of the threats they need to face and the means necessary to cope with them. Moreover, they are usually built with a tight budget and in a short time frame by developers that do not follow security best practices, because they have limited security skills and awareness. As expected, more and more web applications

are being deployed with critical security defects that, when exploited, may have a huge negative effect on the business, potentiated by the worldwide press coverage of such incidents. This is, however, a situation that can be mitigated if developers follow security best practices, since approximately half of the known security vulnerabilities could have been prevented by addressing them at the program code level [1].

For today's organizations, security is becoming more and more a business priority. Thus, they equip their website infrastructure with network firewalls, Secure Sockets Layer (SSL)/Transport Layer Security (TLS), Intrusion Detection System (IDS), Intrusion Prevention Systems (IPS) and Web Application Firewall (WAF). However, the majority of the attacks occur at the *application layer* and these standard network technologies cannot prevent or stop them [2]. To mitigate this problem, a vast number of free and commercial security analysis and testing tools aimed for web application security do exist. To broaden their market, these tools are generic and, obviously, they are not optimized for custom code. This fact negatively affects their results in many situations, as they can easily find some generic problems, but fail on the most serious security issues, which are not so trivial to find and are deeply intertwined into the application's business logic. This limitation results from failing to capture enough knowledge about the inner workings of the web applications, which is critical to allow them detecting more flaws [3]. Existing tools do not only fail to detect all the problems, but they also raise many false alarms, and thoroughly investigating and verifying each one of the potential problems takes time and requires experts with high experience in the security area.

This thesis presents several contributions to tackle the problem of increasing the vulnerability detection capability in web applications. In short, we present a SAST tool to detect two of the most common vulnerabilities, XSS and SQLi, in PHP code. Secondly, we propose an approach to design benchmarks for the evaluation of SAST tools that detect vulnerabilities in web applications. Next, we present three studies combining the results of several SAST tools analyzing different types of PHP source code as a way to increase the vulnerability detection while keeping the number of FP low. Finally, we propose a novel generic hybrid methodology that combines SA, a crawling procedure and DA in order to obtain a set of vulnerabilities that are confirmed to be exploitable, thus maximizing the number of vulnerabilities detected and, at the same time, minimizing the number of FPs.

1.1. Context and Motivation

The World Wide Web (WWW), or simply the Web, was invented in 1989 and is a collection of specially formatted documents stored in servers accessed via the Internet (i.e., a network of networks). The documents are formatted in a markup language called HyperText Markup Language (HTML) that supports links to other documents, as well as other resources, as graphics, audio, and video files. The Web uses the Hypertext Transfer Protocol (HTTP) to transmit data over the Internet. The first-ever website (info.cern.ch) was published on August 6, 1991 by the British physicist Tim Berners-Lee while working at the CERN, in Switzerland. Since then, the number of websites has increased exponentially, and today there are over 1.9 billion websites on the WWW [4]. The website name is a unique hostname that can be converted, using a name

server, into an Internet Protocol (IP) address.

In October 2021, the number of people worldwide using the Internet has grown to 4.88 billion, an increase of 4.8% (222 million new users) compared to October 2020, which is almost 62 percent of the world’s population of 7.87 billion [5]. Cybersecurity Ventures predicts that there will be 6 billion Internet users by 2022 (75% of the projected world population of 8 billion), and more than 7.5 billion Internet users by 2030 (90% of the projected world population of 8.5 billion, 6 years of age or older) [6]. The average Internet user spends 6 hours and 43 minutes online each day. Considering 8 hours a day of sleep, an user is connected to the Internet more than 40% of his life. The rate of growth of Internet connections is outpacing our ability to properly secure it [7].

Global cybercrime costs are on the rise, increasing 15% per cent year over year, according to a 2021 cyberwarfare report by CyberSecurity Ventures [6]. By 2025, it is estimated that cybercrime will annually cost \$10.5 trillion to businesses worldwide. With the global cost of cybercrime at \$3 trillion in 2015, that is more than a threefold increase over a decade. This represents the greatest transfer of economic wealth in history, risks the incentives for innovation and investment, is exponentially larger than the damage inflicted from natural disasters in a year, and will be more profitable than the global trade of all major illegal drugs combined.

The total amount of data created, captured, copied, and consumed globally is forecast to increase rapidly, reaching 77.4 zettabytes in 2021. Over the next four years (up to 2025), global data creation is projected to grow with a compound annual growth rate of 23%, reaching more than 180 zettabytes in 2025 [8]. Cybersecurity Ventures predicts that the total amount of data stored in the Cloud will reach 100 zettabytes by 2025 [6].

The total installed base of Internet of Things (IoT) connected devices worldwide is projected to amount to 30.9 billion units by 2025, a sharp jump from the 13.8 billion units that are estimated in 2021 [9]. Deloitte Global predicts that 320 million consumer health and wellness wearable devices will ship worldwide in 2022. By 2024, that figure will likely reach nearly 440 million units as new offerings hit the market and more health care providers become comfortable with using them [10]. The demand for developing new web applications with increased complexity to feed this ever growing demand in very tight time constraints is tremendous. In fact, there are more than 111 billion lines of new software code being written each year, which introduces a massive number of vulnerabilities that can (and many of them will) be exploited [7].

Web security has been viewed in the context of securing the *web application layer* from attacks by unauthorized users [11]. A **Security Vulnerability** in the web application layer can be attributed either to the use of an inappropriate Software Development Life Cycle (SDLC) model to guide the development process, or to the use of a SDLC model that does not consider security as a key factor [11]. In fact, the behavior of developers has not changed in years. According to a 2013 “Microsoft security study”, 76% of U.S. developers do not use a secure application-program process and more than 40% of the software developers globally said that security was not a top priority for them [12]. In the present days, security still lags and needs to be embedded by default, thus developers need to be aware of security issues and how the software they are developing can be attacked [13].

According to the Open Web Application Security Project (OWASP)¹, an application **security vulnerability** is:

“A hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows an attacker to cause harm to the stakeholders of an application. These stakeholders include the application owner, application users, and others that rely on the application.”

The number of security vulnerabilities in web applications has increased along with the tremendous growth of web applications in last two decades [14]. In spite of all the knowledge available about the security risks involved, this situation seems to be difficult to improve [15] [16]. Tens of thousands of web applications have been developed without any security testing and are exposed to everyone via the Internet [17]. Therefore, a hacker can try to interact with an application in a way that was never expected (i.e., like a software test that was not done during the SDLC, but should have been done to prevent bugs) to find a vulnerability that he could exploit in order to obtain sensitive data or financial assets [18].

CMSs are increasingly used to support the development of web applications, as they provide many built-in features, allowing a rapid application development [19]. Many web applications are built on top of CMS frameworks that can be easily deployed and customized to meet the requirements of a myriad of different scenarios, like personal websites, blogs, social networks, webmail, banking, e-commerce, etc. To cope with this diversity, most CMS-based applications can be extended with third-party services available on the market and new features added through server-side *plugins*, *modules* and *themes* provided by multiple developers.

WordPress is the most popular and widely used CMS adopted by businesses of all sizes and everyday website owners. WordPress is used by 42.2% of all the websites (websites built with or without a CMS), while Shopify, in the second position, is only used by 3.7% of all websites [20]. WordPress has a market share of all CMSs of 65.1%, while the number two has around 4.3%. Below Shopify, there are only four CMSs (Joomla, Squarespace, Wix and Drupal) with more than 1% of all websites. Moreover, in the CMS Usage Distribution in the Top 1 Million Sites, 38.0% are based on WordPress [21]. WordPress and WordPress plugins are developed in the PHP language. PHP is by far the most popular server-side programming language. It powers 78.1% of all web sites [22]. The second most popular is ASP.NET with 8.0% of usage share.

The number of web applications developed using WordPress is huge and there are about 59 thousands of validated third-party plugins in the official WordPress Plugin Directory (WPPD)². These plugins have been downloaded over 1.5 billion times. Furthermore, there are thousands of other third-party plugins available for free or for purchase (e.g., the Envato Market³ has over over 5,900 WordPress plugins available). In practice, developers with basic skills, unknown agendas and uncontrolled software programming practices are implementing new plugins, which ultimately leads to code with suspicious trust levels. The problem becomes even worse as core CMS providers do not run quality assurance procedures on the third-party plugins they use, even for those made available directly on untrusted websites. The only “guarantee” is related with the

¹<https://owasp.org/>

²<https://wordpress.org/plugins>

³<https://codecanyon.net>

comments and ratings of the plugins by their end users and the number of downloads. This is clearly not enough and the reality is that no one can fully trust on the security of the plugins. Not surprisingly, many of them can be exploited by malicious parties, which is the case in many situations [23].

In 2019, over 60% of all installed **CMS** applications were out of date containing vulnerabilities. The percentages of outdated **CMS** installations were 90% for Joomla, 87% for Magento, 77% for Drupal and 49% for WordPress, which is a value lower than for the other popular **CMS** applications. Vulnerabilities continue to come from outdated software plugins, modules, and extensions; abused access control credentials; poorly configured applications and servers; and a lack of knowledge around security best practices. These vulnerabilities are usually found and fixed. However, not all website owners update the software frequently [24].

There are many types of web application vulnerabilities, but the most common are **SQLi** and **XSS**, according to the several statistics presented above. Both types of vulnerabilities are directly attributable to developer errors in the source code, which is the main focus of this thesis, as opposed to other types of errors (for example, errors of Security Misconfiguration (**SM**) can be attributable to system administrators [25], not developers). A **SQLi** attack is a type of code-injection attack that takes data provided by the user and dynamically includes them as part of the structure of the Structured Query Language (**SQL**) code sent to a back-end database [26]. **SQLi** occurs because the user input is not properly validated or is not validated at all. Web applications with this type of vulnerability are extremely exposed to high-risk attacks, as the attacker can potentially extract or modify any data on the database, bypass authentication, or even gain complete access to the underlying databases, computers and networks. On the other hand, a **XSS** attack consists of a malicious injection of script code into a vulnerable web page (by the attacker) that will be later displayed to other users (the victims) without being sanitized or filtered. **XSS** allows attackers to execute scripts in the browser of the victim, to hijack user sessions, steal account credentials, display unwanted advertisements, deface websites, redirect the user to malicious sites, and infect the user with a virus or other malware.

SQLi and **XSS** vulnerabilities occupied the top three spots of the **OWASP** Top 10-2013 list (“The Ten Most Critical Web Application Security Risks”) and Common Weakness Enumeration (**CWE**)/System Administration, Networking and Security (**SANS**) “2019 TOP 25 Most Dangerous Software Errors”⁴. Five years later, in the **OWASP** Top 10-2017 list, **SQLi** vulnerability maintained the first position and **XSS** was moved to the seventh position. In September 2021, a new version was released, **OWASP** Top 10:2021, to reflect changes from 2017 to 2021. In this version, the category A03:2021-Injection (that includes **SQLi**) went down to the third position and the A7:2017-**XSS** became now part of this category [27]. **SQLi** and **XSS** vulnerabilities are prevalent, particularly in legacy code, being the most common attack vectors found in many vulnerability reports⁵⁶⁷.

⁴<https://www.sans.org/top25-software-errors/>

⁵<https://www.veracode.com/security/application-vulnerability>

⁶<https://www.rapid7.com/fundamentals/types-of-attacks/>

⁷<https://it.ucsf.edu/printpdf/884>

1.2. Thesis Contributions

This thesis focuses on **vulnerability detection in web applications (namely WordPress plugins) and proposes techniques and tools for the detection of SQL Injection (SQLi) and Cross-Site Scripting (XSS) vulnerabilities**. As mentioned before, although there are many techniques and tools to detect vulnerabilities, they do not detect all vulnerabilities and they report False Positives (FPs). In practice, we propose new techniques and tools that improve the current situation by detecting more vulnerabilities and reporting less False Positives (FPs). Note that vulnerability detection techniques and tools are especially relevant because they can help developers producing less vulnerable code and security experts identifying security vulnerabilities in production software, thus preventing security breaches. The following provides an overview of the key contributions of the present work:

- 1) A new security analysis methodology and a **SAST** tool, **phpSAFE** for **OOP** web applications (see publication C1), capable of increasing the number of vulnerabilities detected. The tool performs tainted static analysis and is able to analyze PHP code in complex web applications developed within frameworks using **OOP** and based on third-party plugins. The evaluation results against two well-known tools using 35 WordPress plugins showed that the proposed tool clearly outperforms other tools, and that plugins are being shipped with a considerable number of vulnerabilities, which tends to increase over time. The experiments showed that using several tools allows increasing the number of different vulnerabilities detected. Additionally, we also found that many vulnerabilities disclosed to the developers take a long time to be fixed and that many vulnerabilities are not fixed at all. **phpSAFE** can help developers write safer code, earlier in the software development lifecycle.
- 2) A new benchmarking methodology for **SAST** tools for web security (see publication J2). We propose a general approach to design benchmarks for the evaluation of **SAST** tools able to detect software vulnerabilities, considering workloads characterized in terms of vulnerable and non-vulnerable lines of code and including real vulnerable applications representative of scenarios with different levels of criticality, and different ranking metrics.
- 3) A benchmarking campaign of **SAST** tools for web security (see publication C2). We present a concrete instance of the general benchmark methodology to demonstrate its feasibility, evaluating five **SAST** tools for the detection of **SQLi** and **XSS** vulnerabilities in a workload composed of 134 WordPress plugins organized in four scenarios of different criticality and using specific evaluation metrics tuned for each scenario. We also describe a comparative evaluation of the ranking obtained using our instantiation of the benchmark with the Software Assurance Metrics and Tool Evaluation (**SAMATE**) methodology and the Benchmark for Security Automation (**BSA**) of **OWASP**.
- 4) Case studies on combining the results of diverse **SAST** tools. Developers frequently rely on free **SAST** tools to automatically detect vulnerabilities in the source code of their applications, but it is well-known that the performance of such tools is limited and varies from one software development scenario to another, both in terms of the rate of vulnerabilities found and **FPs** reported. Diversity is an obvious direction to take to improve coverage, as different tools usually report distinct vulnerabilities, but this may come with an increase in the number of false alarms. We conducted several case studies combining the

results of five **SAST** tools searching for **SQLi** and **XSS** vulnerabilities, considering three adjudication strategies: *1-out-of-n*, *n-out-of-n* and *majority voting*. Our findings revealed that the best solution depends on the development scenario. Furthermore, an in-depth analysis of the code of several WordPress plugins provided insights on why some tools do not detect vulnerabilities that other tools detect.

- 5) A new methodology blending static and dynamic analysis for web application vulnerability detection. The methodology combines **SA**, a crawling procedure and **DA** into a number of steps that can be fully automated, in order to obtain a set of vulnerabilities that are confirmed to be exploitable. In short, it starts by performing a **SA** to produce a list of candidate vulnerabilities. Next, the application is executed automatically, stopping only when the code where the vulnerabilities are located is run. A set of specific inputs and configuration options are automatically generated from the results of the **SA** and the runtime information collected, which are used to guide the **DA** in the process of successfully exploiting each vulnerability reported by the **SA**. Unlike other approaches, our methodology has the ability to automatically generate the necessary data to feed the **DA**, so it is able to go through the execution path where the vulnerability is located and trigger it, therefore increasing the number of vulnerabilities that can be confirmed by the **DA**.
- 6) A case study using the proposed blending of static and dynamic analysis for web application vulnerability detection. We created an instantiation of the methodology to demonstrate its feasibility, using more than 450 **SQLi** vulnerabilities in 49 WordPress plugins. Our approach was able to confirm either as a vulnerability or a false alarm 76.7% of the results reported by the **SA**, decreasing tremendously the usual need for manual analysis, which is a huge improvement for security practitioners.

1.3. Structure of the Thesis

This document comprises six chapters apart from the introduction.

Chapter 2. After the initial motivation and problem statement have been described, the state of the art is highlighted in order to bring the thesis contribution into context. It includes background and related work on web applications and vulnerabilities, vulnerability detection using **SA** and **DA**, and benchmarking of **SAST** tools.

Chapter 3 presents the **phpSAFE SAST** tool for detecting **SQLi** and **XSS** vulnerabilities in PHP code. We discuss in detail the architecture of the tool and its instantiation for WordPress plugins. Details about the evaluation carried out are also presented.

Chapter 4 proposes an approach for benchmarking **SAST** tools for web security. The chapter also presents an instantiation of the benchmarking approach for **SQLi** and **XSS** vulnerabilities using a set of 134 WordPress plugins and five **SAST** tools.

Chapter 5 describes case studies combining the results of several **SAST** tools considering three configurations for the adjudicator as a way to increase the number of vulnerabilities detected. This includes an in-depth analysis of the code of several plugins for highlighting the strengths and weaknesses of the **SAST** tools.

Chapter 6 proposes a generic methodology that combines [SA](#) and [DA](#) to effectively increase the number of reported vulnerabilities while reducing the number of [FPs](#). An instantiation of the methodology to demonstrate its feasibility, using a large number of [SQLi](#) vulnerabilities in WordPress plugins, is also discussed.

Chapter 7 concludes the thesis and proposes topics for future research.

The document includes a number of appendixes with complementary material. **Appendix A** provides details about the stage of “[Assigning Applications to Scenarios](#)” of Section [4.1.3.2](#). A complete list of WordPress plugins that compose the workload created in Section [4.2](#) is included in **Appendix B**. **Appendix C** offers details about the “[Experimental Evaluation](#)” of the benchmark instantiation presented in Section [4.3](#). **Appendix D** details the results of the case study presented in Section [5.1](#). **Appendix E** presents an instantiation of the benchmarking approach described in Section [4.2](#) for [SQLi](#) and [XSS](#) vulnerabilities using synthetic test cases.

Background and Related Work

This chapter provides an overview of the state-of-the-art in the fundamental area this work, as well as in related research areas. It starts by overviewing essential concepts regarding computer security and information security. Then, it presents background on web applications and on their most common vulnerabilities, followed by existing security technologies that can be applied. Next, it discusses the relevant work related with vulnerability detection through static, dynamic and hybrid analysis. Finally, it addresses the concept of benchmarking security tools for static analysis.

2.1. Computer Security Concepts

The modern society is increasingly dependent on the Internet. It impacts our personal lives, our businesses and our essential services. Security embraces individual, companies and governments and covers a broad range of issues related to financial health, national security, whether through terrorism, crime and industrial espionage. **Cybercrime** is a fast-growing area of crime focusing on theft, financial crimes, corruption, abuse (especially crimes against children), hacking or denial of service to vital systems. The risk of industrial cyber espionage, in which one company makes active attacks on another, through Internet, to acquire high value information is also very real [28].

Security is fundamentally about protecting **assets**. Assets may be tangible items, such as operations or costumers database, but they may also be less tangible, such as the reputation of a company. For a financial services company, the asset might be client information or other data that are used in transactions. The value of assets are directly related to the value of the business transactions that it supports. Governments, military, corporations, financial institutions, hospitals and private businesses accumulate a great deal of confidential information about their

employees, products, customers, partners, research and financial status. Most of this sensitive information is actually collected, processed and stored on computers and transmitted across networks to other computers. The value of this information is difficult to evaluate. Cybersecurity Ventures predicts cybercrime to inflict damages totaling \$6 trillion USD globally in 2021 [6].

In the current context, it very important for the organizations to define requirements of security for their information systems and to identify potential threats across key business areas, including people, processes, data, and technology throughout the entire organization. The threats identified must be addressed according to its degree of risk, thus minimizing the risk and its associated costs. Developing a secure information system includes knowing the threats that it will be exposed, making effective trade-offs, and integrating security throughout the **SDLC**. Organizations should build information systems that are secure by design, meaning that security is intrinsic to their business processes, product development, and daily operations. Security should be factored into the initial design, not bolted on afterward [29].

Security of a system is a combination of its ability to support system availability, data integrity and data confidentiality. A failure of a system to protect any of these properties results in a security violation [30]. The National Institute of Standards and Technology (**NIST**) Glossary of Key Information Security Terms - Revision 2 (2013) provides the following definitions for *Computer Security* [31]:

Measures and controls that ensure *confidentiality, integrity, and availability* of information system assets including hardware, software, firmware, and information being processed, stored, and communicated.

The term *Information Security*, sometimes shortened to **InfoSec**, is actually used with the idea of Computer Security. The **NIST** Glossary provides the following definition for **Information Security**, which also refers to the Confidentiality, Integrity and Availability (**CIA**) [31]:

The protection of information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide confidentiality, integrity, and availability.

The definitions of *Computer Security* and *Information Security* introduce three fundamental concepts that are considered the principles of information security. These principles have also been called security goals, objectives, properties or pillars. More commonly, they are known as the **CIA Triad**:

- 1) **Confidentiality**: is the concealment of information. This property ensures that private or confidential information is not accessed, used, copied, made available, or disclosed by anyone except the authorized individuals. A loss of confidentiality is the unauthorized disclosure of information.
- 2) **Integrity**: is the assurance that the information is trustworthy and accurate. It assures that the information is not created, changed, or deleted by unauthorized individuals in a way that is not detectable by authorized users. A loss of integrity is the unauthorized modification or destruction of information.
- 3) **Availability**: refers to the ability to use the information. It guarantees that the information

is accessible for use by authorized users. Authorized users should be able to access data whenever they need to do so. A loss of availability is the disruption of access to or use of information.

The *CIA triad* is a fundamental security model that has been around for more than 20 years. It serves as a tool or guide for securing information systems and networks and related technological assets. Figure 2.1 presents a graphical presentation of how the *CIA* triad effectively protects data. Having the data triangle in the middle, further re-enforces the point that all three components must be protected in order to comply with the *CIA*. It also stresses the importance to balance the relationships among the components in the triad.

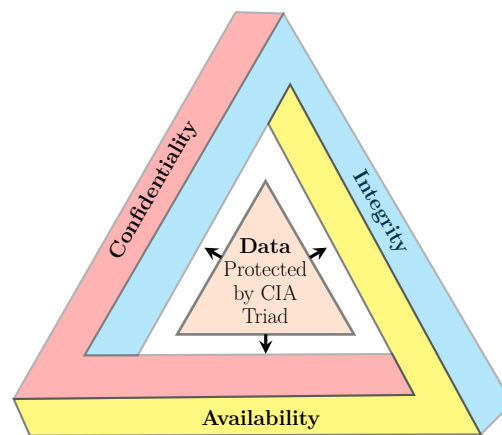


Figure 2.1.: The CIA Security Triad: Confidentiality, Integrity, Availability.

There is a continuous debate about extending the classic *CIA* triad. Issues such as **non-repudiation**, **possession** and **utility** do not fit well within the three core concepts. **Legality** is also becoming a key consideration for practical security installations. The **Parkerian Hexad** is a set of six elements of information security that adds three additional properties to the three classic security properties of the *CIA* triad, illustrated in Figure 2.2. These properties are the following [32]:

- 1) **Non-Repudiation or Authenticity:** the property of being genuine and being able to be verified and trusted. Authenticity refers to the guarantee that the information is correctly characterized according to what is in fact, it is not fraudulent. For example, the correct attribution of origin such as the authorship of an e-mail message or the correct description of information such as a data field that is properly named. Non-repudiation means that once an action has taken place, a user cannot realistically claim that he did not make it. Thus, non-repudiation guarantees the protection against an individual falsely denying having performed a particular action.
- 2) **Possession or Control:** this property guarantees that the information is only under control of the authorized individuals. The ownership or control of information, which is distinct from confidentiality. For example, if confidential information, such as a user ID and password combination, is in a sealed container and the container is stolen, the owner justifiably feels that there has been a breach of security even if the container remains closed. This is a breach of possession or control over the information.
- 3) **Utility or Usefulness:** this property is related with the guarantee that the information

is usefulness for its particular purpose. For example, if information are encrypted and the decryption key is unavailable, the breach of security is in the lack of utility of the information. This data are still confidential, possessed, integral, authentic and available.

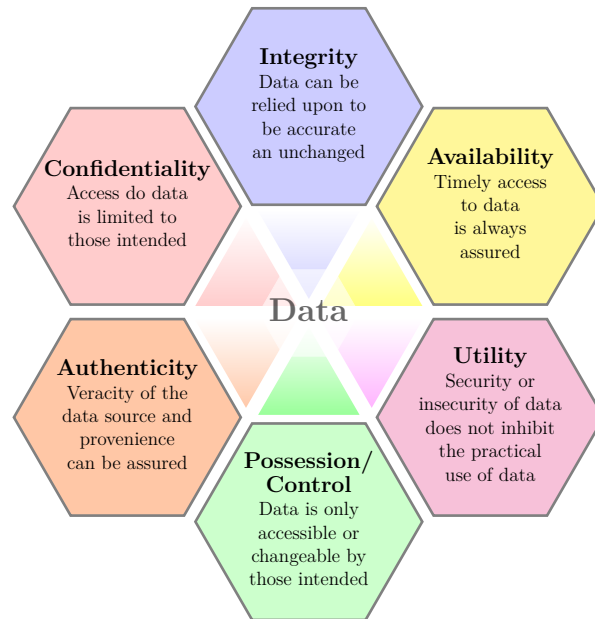


Figure 2.2.: The Parkerian hexad security requirements (adapted from [33]).

Each of the six fundamental, atomic and non-overlapping properties can be violated independently of the others, with one important exception: A violation of confidentiality always results in loss of exclusive possession, at the least. Loss of possession does not necessarily result in loss of confidentiality. In practice, a system is secure if its resources are used and accessed as intended under all circumstances. Unfortunately, total security cannot be achieved. Once a machine is plugged in, there are practically an infinite number of ways its use might deviate from its intended purpose. This deviation is a malfunction. When the difference between the expected behavior and actual behavior is caused by an adversary (as opposed to simple error or accident), then the malfunction is a “security” problem.

An **asset** is a resource of value such as the data in a database or on the file system related to the web application that must be protected [34]. Personal data, health information, intellectual property, and access to critical operations are all assets. For example, credit card numbers have been the prime target for thefts and breaches. Thus, user credit card numbers are an asset that must be protected very carefully in environment of the web application. The next paragraphs introduce technical terms regarding vulnerabilities, threats due to their existence and countermeasures to prevent their exploitation.

Threat is any circumstance or event with the potential to adversely impact organizational assets and operations through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service [35]. In other words, a threat is something bad that can happen to an asset. For example, the injection of malicious code (**SQLi** attack) in the pair username-ID/password of any web application login form in an intended attempt to manipulate the database in some way. Technical impacts include loss of confidentiality,

integrity and availability. A common business impacts are financial and reputation damage, non-compliance and privacy violation.

Threat agent (or threat source) is the intent and method targeted at the intentional exploitation of a vulnerability or a situation and method that may accidentally trigger a vulnerability [36]. Attackers can potentially use many different paths through any vulnerable web application to do harm to the business or organization. Each of these paths represents a **risk** that may, or may not, be serious enough to warrant attention.

Vulnerability is a weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source [37]. Therefore, vulnerabilities are weaknesses in web applications that guards or operates on sensitive data. Vulnerabilities can be leveraged to force software to act in ways it is not intended to, such as gleaning information about the current security defenses in place. Fault to properly validate data at entry and exit points of the web application are examples of vulnerabilities in web applications, which can result in input validation attacks.

Attack vector is a method or a pathway by which a hacker can gain access to a computer or network server in order to deliver a payload or malicious outcome [38]. Examples of attack vectors are: malicious email, pop-ups, instant messages, text messages, attachments, worms, web pages, downloads, deception (aka social engineering), hackers. Examples of payloads are: viruses, spyware, Trojans, malicious scripting/executable. Hackers steal information, data and money from people and organizations by investigating known attack vectors and attempting to exploit vulnerabilities to gain access to the desired system.

An **attack** (or exploit) is an intended action to harm an asset by exploiting vulnerabilities. It makes effective a threat. Examples of attacks include sending malicious input to a web application, or flooding a network in an attempt to deny service. The injection of malicious input such as code, scripts, commands, that can be interpreted and/or executed by different targets to exploit vulnerabilities in web browsers (e.g., [XSS](#)), database servers (e.g., [SQLi](#)), server side file processing (eXtensible Markup Language ([XML](#)) and XML Path Language ([XPath](#)) Injection) and operating systems (e.g., Unrestricted File Upload ([UFU](#)) and Buffer Overflow ([BO](#))).

Risk is defined as the potential for loss or damage when a threat exploits a vulnerability [39]. Examples of risk include: financial losses, loss of privacy, reputation damage, legal implications, even loss of life. Risk can also be defined as: $Risk = Threat \times Vulnerability$.

Countermeasures (security controls or safeguards) are actions, procedures, techniques, or other measures that oppose a vulnerability of an information system by eliminating or preventing it [40]. The goal of a countermeasure is to minimizing or to eliminate the harm that it can cause.

To summarize, a **threat** is a potential event, which a **threat agent** can intentionally operate through an **attack** to exploit a **vulnerability** in an attempt to adversely affect an **asset** related with a web application. The success of an attack may results in technical and business impacts. The level of impact or severity of each risk should be evaluated in order to prioritize **countermeasures** to addresses a threat and mitigates the **risk** [15].

Figure 2.3, illustrates many possible paths to harm assets or functions with consequent technical

and business impacts. These paths, sometimes are trivial to find and exploit and sometimes they are extremely difficult. Similarly, the harm that is caused may be of no consequence, or it may lead to total system compromise. To measure the risk of an organization related with an web application, may requires an approach (e.g., [OWASP Risk Rating Methodology](#)¹) that evaluate the likelihood associated with each threat agent, attack vector, and security weakness and combine it to estimate the technical and business impact for the organization [41].

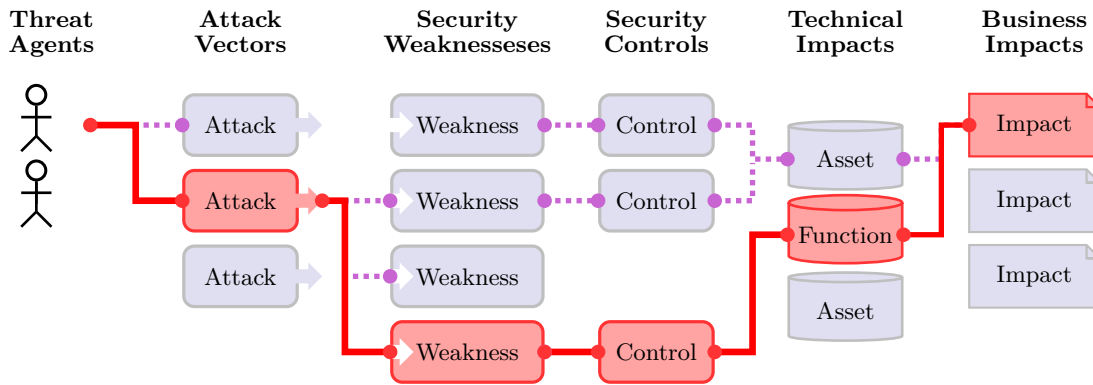


Figure 2.3.: Application Security Risks (adapted from [41].)

2.2. Web Applications and Security

A web application is a kind of *client-server* application that runs on default web browser of any device such as a desktop computer, laptop, tablet, smartphone, smart TV, among others. This is a tremendous advantage of web applications over desktop applications, because they perform their function independently of operating systems and **web browsers** running on the client side. Web browsers are software applications that allow users to retrieve data and interact with content located on hypertext documents within a **website**. Another significant advantage of web applications is that they can quickly be deployed anywhere and instantly be accessible by millions of users around the globe at virtually no cost and without any installation requirements at the user's end.

As shown in Figure 2.4, a web application consists of code on both the *server-side* (actually providing the service) and the *client-side* (the web browser of the user accessing the service). The server-side includes a *web server*, *web application server*, *database server* and *interpreters* or *runtimes* such as PHP Hypertext Preprocessor ([PHP](#)), Java Platform and .NET Framework. The communication between the client-side and the server-side is performed using the protocol [HTTP](#)². It is the foundation of data communication for the **World Wide Web**, where web documents include hyperlinks to other resources that the user can easily access, for example by a mouse click or by tapping the screen in a web browser. The [HTTP](#) protocol provides functions as a *request-response* protocol in the *client-server* computing model³. In [HTTP](#), a web browser,

¹https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

²HTTP is an application layer protocol for distributed, collaborative, hypermedia information systems.

³https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

for example, may be the client and an application running on a computer hosting a website may be the server.

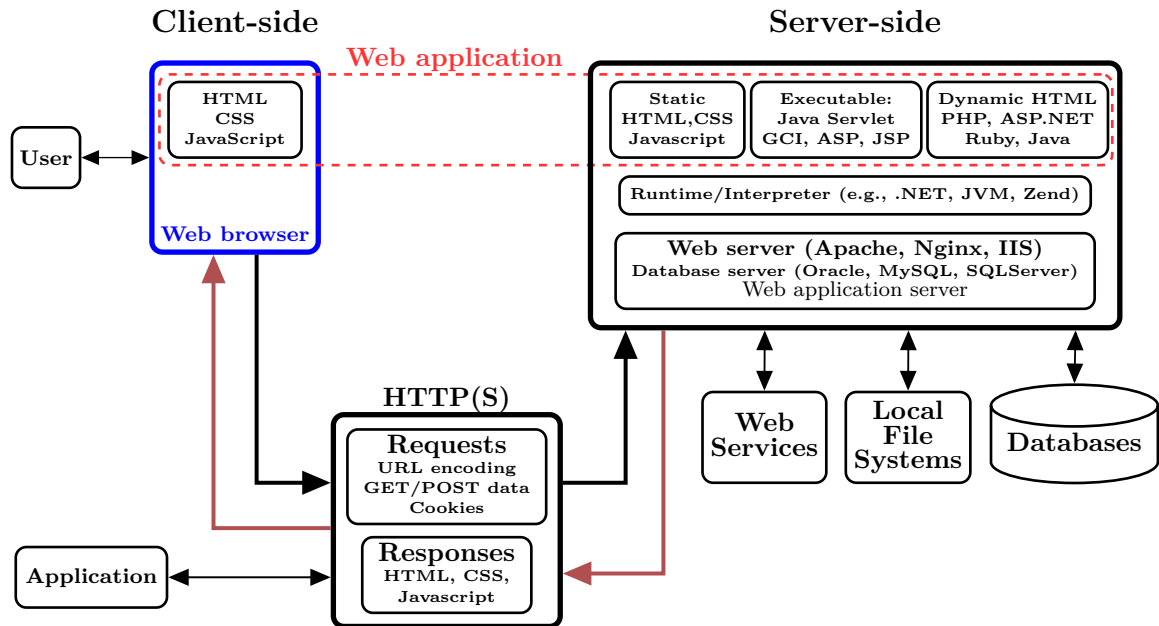


Figure 2.4.: Overview of web application (adapted from [42]).

On the *server-side*, the web application receives user inputs via [HTTP](#) requests from the browser of the user and interacts with local file systems, back-end databases, or other components for data access, such as *Web Services*. The web application dynamically generates a *web document*⁴ according the inputs of the user in a standard format (i.e., in [HTML](#) using Cascading Style Sheets ([CSS](#)) and [JavaScript](#)) to allow support by all browsers and the web server sends this document to the browser of the user through [HTTP](#) responses.

On the *client-side*, web documents are rendered into web pages with embedded client-side code (e.g., [JavaScript](#)). This code can be interpreted and executed in the web browser of the user whenever the page is rendered, or according to interactions of the user. Client-side code is used to program the behavior of web pages and create dynamic [HTML](#). It can access, change, add and remove all the [HTML](#) elements, attributes and events in the web page. Moreover, the client-side code can also communicate with the server-side code asynchronously, without interfering with the display of the existing [HTML](#) page via Asynchronous Javascript and XML ([AJAX](#)), and dynamically updates parts of the web page, without the need to reload the whole page. This technology makes it possible for web applications to have a rich interface and a user experience that resembles the desktop counterpart, making it even easier for everyone to work with them.

Web applications are often developed with insufficient security requirements, ignoring most of the threats they face and the means necessary to cope with them. Moreover, they are usually built with a tight budget and short time frame by developers that do not follow security best practices. In fact, developers have a huge pressure in fulfilling the customer's requirements and

⁴Web document is often defined as a simple [HTML](#) file. A web document is often made up of several files (e.g., images) and is accessed via a URL. Web documents are formatted in a markup language like [HTML](#) or [XML](#) and can also contain content produced with scripting languages like [JavaScript](#), [PHP](#), and [Perl](#).

switch to the next project as quickly as possible. Due to the pressure in developing more and more web applications, there is a huge demand for web developers, so many of them have limited security skills and awareness knowledge. As expected, more and more web applications are being deployed to the public with critical security defects that, when exploited, have a tremendous effect on the business, potentiated by the worldwide press coverage of such incidents.

As the quality of the code of many web applications is often rather poor and many vulnerabilities of commonly used code are published [43], an ever-increasing number of attacks target web applications. According to a Gartner study, 75% of all attacks on websites target the application level and not the infrastructure [44]. These attacks are performed through the default open ports (e.g., 80 for **HTTP**, 443 for Hypertext Transfer Protocol Secure (**HTTPS**)) used for legitimated traffic. Traditional network perimeter security methods cannot stop these attacks, because they use legitimate channels with well-formatted **HTTP** packets and web applications are, by nature, designed to allow visitors to access data in websites. By exploiting simple vulnerabilities in web applications, an attacker can pass through perimeter security undetected, accessing data even with traditional network firewalls and **IDS** systems in place and well configured [45]. To stop these attacks, the security mechanisms must be aware of the application logic and they should perform deep packet inspection, which increases significantly their complexity and poses new challenges for their development.

Several techniques (countermeasures) exist to develop secure web applications [46]. These techniques can be categorized into three categories, according to the phase in which they occur in the **SDLC** of the application (from development, review and auditing to deployment) [42]:

- 1) **Secure construction.** Developers must follow security code guidelines to build secure software [47]. Usually, they should follow the best coding practices and use secure libraries (when available for the programming language). Developers should also be trained to learn on how to program with security in mind. However, this technique does not guarantee the development of software free of weaknesses, because most developers either do not follow secure code guidelines or repeat the same type of programming mistakes in their code [47] [48].
- 2) **Security analysis and testing.** These techniques are based on program analysis to identify vulnerabilities within web applications during the coding phase of the **SDLC**. In the literature, vulnerability detection approaches are commonly divided into three wide classes [47]:
 - a) **Static Analysis (SA):** or white-box analysis, such as source code review.
 - b) **Dynamic Analysis (DA):** or black-box analysis, such as penetration testing.
 - c) **Hybrid Analysis (HA):** or gray-box analysis, which is a combination of white-box analysis and black-box analysis.
- 3) **Runtime protection.** This approach is focused on protecting a potentially vulnerable web application, preventing external exploits while the application is running. These techniques need a runtime environment, like a proxy, and can be divided into two classes [42]:
 - a) **Taint-based protection:** pre-processes the source code of the application in order to identify untrusted user inputs.
 - b) **Taint-free protection:** aims to directly, without tracking user input, detect the

input validation attack before it even reaches the web application or after it triggers a vulnerability in the application. This class of techniques usually require an additional phase to establish detection models.

2.3. Common Web Application Vulnerabilities

The number of the possible ways a web application may be vulnerable is quite vast, but most of the problems fall into a limited number of situations. The [OWASP Top 10](#) is a standard awareness document for developers and web application security, it contain a list of the ten most critical web application security risks, along with effective methods of dealing with them [41]. It represents a broad consensus about the most critical security risks to web applications. The [OWASP](#) encourages people to use the Top 10 to get organizations started with application security. Developers can learn from the mistakes of other organizations. Executives should start thinking about how to manage the risk that software applications create in their enterprise.

Table 2.1.: OWASP Top 10 most critical web application security risks (2017 and 2021).

OWASP Top 10 - 2017	⇒	OWASP Top 10 - 2021
A1:2017 - Injection	↘ ³	A01:2021 - Broken Access Control
A2:2017 - Broken Authentication	↘ ⁷	A02:2021 - Cryptographic Failures
A3:2017 - Sensitive Data Exposure	↗ ₂	A03:2021 - Injection
A4:2017 - XML External Entities (XXE)	↘ ⁵	A04:2021 - Insecure Design ^(New)
A5:2017 - Broken Access Control	↗ ₁	A05:2021 - Security Misconfiguration
A6:2017 - Security Misconfiguration	↗ ₅	A06:2021 - Vulnerable and Outdated Components
A7:2017 - Cross-Site Scripting (XSS)	↗ ₃	A07:2021 - Identification and Authentication Failures
A8:2017 - Insecure Deserialization	→	A08:2021 - Software and Data Integrity Failures ^(New)
A9:2017 - Using Components with Known Vulnerabilities	↗ ₆	A09:2021 - Security Logging and Monitoring Failures
A10:2017 - Insufficient Logging & Monitoring	↗ ₉	A10:2021 - Server Side Request Forgery (SSRF) ^(New)

The [OWASP Top 10](#) was first released in 2003, minor updates were made in 2004 and 2007. The [OWASP Top 10 2010](#) and 2013 versions were revamped to prioritize by risk, not just prevalence [49]. Change has accelerated over the last years. To keep up with the changes between 2013 and 2017 the [OWASP Top 10](#) changed to reflect these changes. In September 2021, was released a new version, [OWASP Top 10:2021](#), to reflect changes from 2017 to 2021. In this version, the category *A03:2021-Injection* slides down to the third position and the *A7:2017 - Cross-Site Scripting (XSS)* is now part of this category. Table 2.1 lists the [OWASP Top 10](#) most critical web application security risks for versions of 2017 and 2021 and what changed from 2017 to 2021. In summary, there are three new categories, four categories with naming and scoping changes, and some consolidation in the Top 10 for 2021. Names have changed to focus on the root cause over the symptom [27].

Almost every web application depend on a database server, operating system features and external programs, such as `Sendmail`. An injection occurs when an application passes untrusted

data (e.g., user input) without any validation to another system for execution. Thus, an attacker can inject special characters, malicious commands, or command modifiers into the data to execute malicious actions. These include, calls to the operating system through system calls, use of external programs with shell commands and calls to back-end databases through [SQL](#). Injection vulnerabilities are often found in [SQL](#), Lightweight Directory Access Protocol ([LDAP](#)), [XPath](#), or NoSQL queries, OS Commands ([OSCs](#)), [XML](#) parsers, Simple Mail Transfer Protocol ([SMTP](#)) headers, expression languages, and Object Relational Mapper ([ORM](#)) queries. Injection has topped the [OWASP Top 10](#) since around 2010 [15].

[XSS](#) and [SQLi](#) are not only two of the most common vulnerabilities found in web applications, but they are also widely exploited by hackers and the organized crime [50] [51] [42], [52], [53]. These vulnerabilities are injection vulnerabilities mainly caused by poor or lack of validation of the web application input values. Due to their relevance in the web security scenario, they are the main focus of the present work and they are detailed in the following sections.

2.3.1. Cross-Site Scripting

Cross-Site Scripting ([XSS](#)) is the most prevalent vulnerability in the 2020 [CWE Top 25 Most Dangerous Software Weaknesses](#) [54] and is the seventh most prevalent vulnerability in the [OWASP Top 10 - 2017](#). It is found in around two thirds of all applications [15].

A [XSS](#) vulnerability occurs whenever data enters a web application through an untrusted Entry Point ([EP](#)) (e.g., user input) and the data is included in dynamic content sent to the browser of the user without being validated for malicious content [47]. The malicious content often takes the form of a segment of [JavaScript](#), [VBScript](#), [ActiveX](#), [HTML](#), [Flash](#) or any other type of code that the browser may execute into vulnerable pages to fool the user, executing the script in the environment of his web browser.

[XSS](#) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. The victim is not attacked directly. Instead, the attacker injects the malicious code through a vulnerability of the web application that the victim visits. The browser of the victim cannot distinguish between the legitimate code of the web application and the malicious code injected. Unintentionally, the web application acts as an accomplice of the attacker delivering the malicious code for him.

The scripts executed in a web browser run in a very restricted environment that has extremely limited access to the files of the user and operating system. However, within the environment of the web browser, the script code controls the behavior of web pages within the browser. Thus, considering the following facts, the scripts can be dangerous [55], for example:

- Script code has access to some of the sensitive information of the user contained in [JavaScript](#) such as `document.cookie` property, `window.history`, `window.location` and `window.navigator` objects.
- Script code may contain [AJAX](#). The script can use the `XMLHttpRequest` object to send [HTTP](#) requests with arbitrary content to arbitrary destinations.
- Script code can make arbitrary modifications to the [HTML](#), [CSS](#), and [JavaScript](#) of the

current page by using Document Object Model (DOM) manipulation methods. It can be used for defacing web pages or inserting links to redirect the user to dangerous websites.

- **JavaScript** in modern browsers can use HTML5 Application Programming Interfaces (APIs). For example, it can gain access to the geolocation of the user, webcam, microphone, and even specific files from the file system of the user. Most of these APIs require user opt-in, but the attacker can use social engineering⁵ to go around that limitation.

These facts open many possibilities for the development of malicious scripts that can cause security breaches. When executed in the browser of the user, XSS allows an attacker to perform many types of attacks, from which the following are the most common [56]:

- **Phishing.** The attacker, in an attempt to acquire sensitive information such as usernames and passwords, can insert a fake login form into a trustworthy page using DOM manipulation methods, set the `action` attribute of a web form to target a web server controlled by him. When the user signs in, the sensitive data are stored on the web server of the attacker in persistent storage like a text file or a database.
- **Cookie theft.** The attacker can access the cookies of the victim associated with the website using the `document.cookie` property, send them to his own server, and use them to extract sensitive information like session IDs. With this data the attacker can hijack the current session of the victim and impersonate him.
- **Keylogging.** Using the JavaScript `addEventListener` method, the attacker can register a keyboard event listener (e.g., `keyup`) and then send all of the keystrokes of the user to his own web server, potentially recording sensitive information such as usernames, passwords, credit card numbers and associated data.

Figure 2.5 shows a diagram with a general pattern of a typical XSS attack. First the hacker infects a legitimate web page with his malicious client-side script. When a user (victim) is tricked to visit this web page the script is downloaded to his browser and executed [57]. The XSS attack occurs every time that the exploited web page is loaded into the browser of any user. For the user, it is difficult to detect this type of attacks because the attacker has injected the malicious code into a web page served by a (likely to be) trustworthy website. Thus, in the Uniform Resource Locator (URL) bar of the browser it is shown the name of the website, and the malicious code is considered a legitimate part of the website and it is executed in the context of the website.

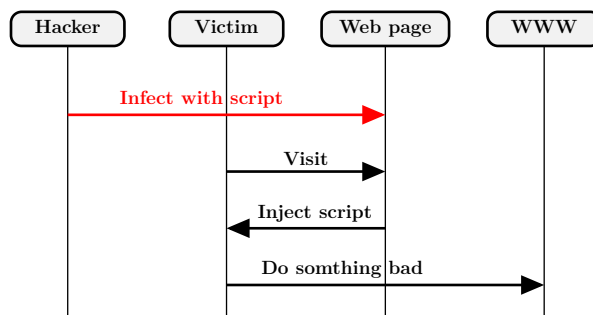


Figure 2.5.: A high level view of a typical XSS Attack.

⁵Social engineering - psychological manipulation of users into performing actions.

XSS vulnerabilities are often divided into three different types (Reflected XSS, Stored XSS, DOM based XSS), as discussed in the following subsections (the last subsection discusses how to prevent XSS vulnerabilities).

2.3.1.1. Reflected XSS Vulnerabilities

In reflected XSS vulnerabilities the malicious script code is part of the request of the user to the web site. This code is dynamically included (reflected) in the response send back to the user, as part of the web page response, without any validation or escaping [58].

Reflected XSS vulnerabilities can be exploited, for example, by sending a fake email that mimic the image of a well-known and reliable company in order to draw the attention of victims. Typically, the contents of the e-mails promise extravagant promotions for the user or request to do an update of your bank details, avoiding the cancellation of the account. The e-mail also contains, a specially-crafted link that points to the trustworthy website. Less attentive users are tricked to click in these link and the attack occurs.

Consider the following attack scenario adapted from [56]: an attacker exploit a reflected XSS vulnerability in a legitimate website with the goal to steal the cookies of the victim. Figure 2.7 illustrates how this attack example can be performed by an attacker with the following steps:

- 1) The attacker crafts a URL containing a malicious string and sends it to the victim.
- 2) Using social engineering skills, the victim is tricked by the attacker into clicking on the malicious URL.
- 3) Unintentionally, the website includes the malicious string present in the URL in the response page sent to the victim.
- 4) The browser of the victim executes the malicious script it received inside the response. The script orders the browser to send the website's cookies stored in the victim's browser to the server of the attacker.

In the example, the vulnerability is locate at the line 4 of the PHP code fragment in Listing 2.1. In PHP language the keyword `$_GET` is an associative array of variables passed to the current script via the URL parameters [59]. In this example, it is used to capture the content of the search keyword (`keyword`) that comes from the user, crafted by the attacker. It contains the malicious script which is sent back (reflected) with the `echo` PHP language construct to the browser of the victim where it is executed as a legitimate script.

2.3.1.2. Stored XSS Vulnerabilities

Stored XSS is a type of vulnerability where the malicious string is previously stored by the attacker in the database of the website. The script is only executed when the victims access the web pages that serve back the malicious string. Attackers often try to get more victims to visit the vulnerable web page so they send spam messages or promote the page on social networks. The diagram of the Figure 2.6 illustrates how this type of attack can be performed:

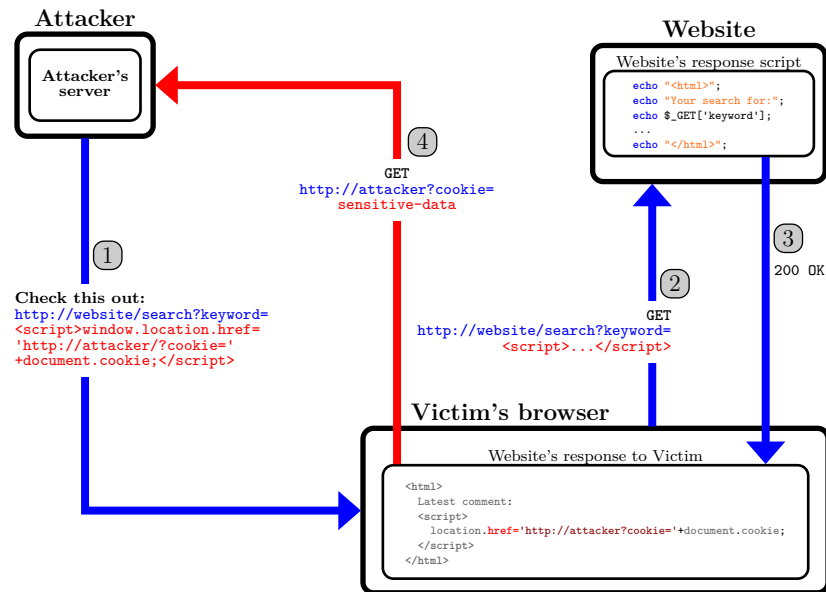


Figure 2.6.: Reflected XSS attack to steal browser cookies. Adapted from [56].

```

1 <?php
2 echo "<html>";
3 echo "Your search for:";
4 echo $_GET['keyword'];
5 ...
6 echo "</html>";
7 ?>

```

Listing 2.1: Example of code with XSS vulnerability.

- 1) The attacker uses one text field inside a form of the website to insert a malicious string into the database of the website.
- 2) The victim requests a page from the website that is filled with information coming from the databases.
- 3) The website includes the malicious string from the database in the HTTP response and sends it to the victim.
- 4) The browser of the victim executes the malicious script inside the response, sending the cookies of the victim to the server of the attacker.

In the scenario of this stored XSS vulnerability, the attacker can easily corrupt a database using a legitimate form of the website with malicious string. Later, each time a user visits this website, the malicious string will be sent and executed on his web browser. Thus, stored XSS vulnerabilities are much more damaging than reflected XSS vulnerabilities, for two reasons [60]: they do not require much social engineering, as the attacker can directly supply the malicious string without tricking users into clicking on a URL; and a single malicious string planted once into a database executes on the browsers of many victim users.

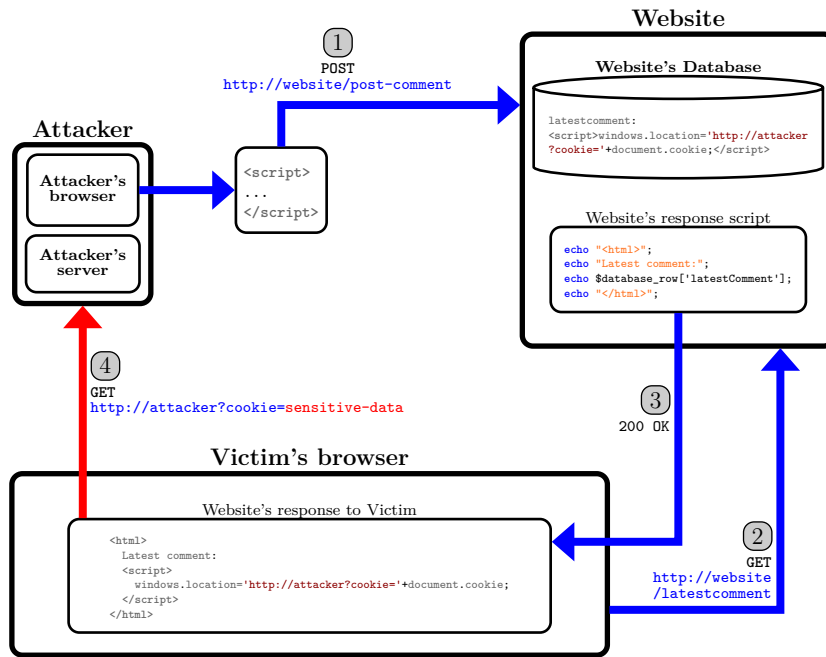


Figure 2.7.: Stored XSS attack to steal browser cookies. Adapted from [56].

2.3.1.3. DOM-Based XSS Vulnerabilities

DOM-based XSS vulnerabilities is very similar to reflected XSS vulnerabilities but, in this case, the malicious string is injected in the client-side rather than in the server-side. Thus, it does not need that any information to be sent or echoed by the web server to exploit the vulnerability [61], therefore bypassing any security mechanisms that may be present in the web server. Figure 2.8 shows a diagram to illustrate how DOM-based XSS attack works, adapted from [56]:

- 1) The attacker crafts a URL containing a malicious string and sends it to the victim.
- 2) The victim is tricked by the attacker into requesting the URL from the website.
- 3) The website receives the request, but does not include the malicious string in the response.
- 4) The browser of the victim executes the legitimate script inside the response, causing the malicious script to be inserted into the page.
- 5) The browser of the victim executes the malicious script inserted into the page, sending the cookies of the victim to the server of the attacker.

In the example of the DOM-based XSS attack, the server do not inserts the malicious script into the response page to the victim. Thus, the server-code is free of vulnerabilities. The lines 4 and 5 of legitimate script code directly make use of the user input in order to add dynamically HTML to the page being loaded. The lines are list below:

```
4 var keyword = location.href.substring(9); // Extracts characters from position 9 to the end of the URL.
5 document.querySelector("em").innerHTML = keyword; // Injects the malicious script in the <em> element.
```

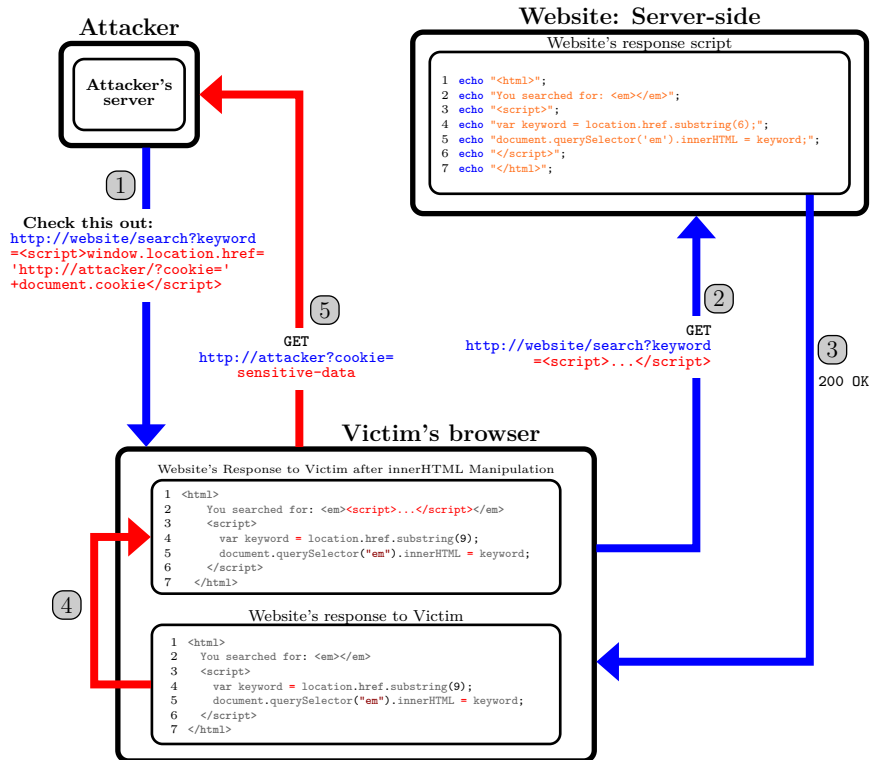


Figure 2.8.: High level of a DOM based XSS attack. Adapted from [56].

In line 4 the variable `keyword` stores the value of the user data provided in the query string of the URL of the website, ignoring the first 8 characters that corresponds to the symbol “?” plus the name of the parameter of the URL (`keyword`) plus the symbol “=”. Thus, the variable `keyword` stores the malicious script:

```
<script>windows.location='http://attacker/?cookie='+document.cookie</script>
```

In the second line, the malicious script is injected inside the HTML element `` of the current page by using the `document.querySelector`⁶ method to get a reference to the first HTML `` element within the document and the DOM manipulation `innerHTML`⁷ property to set HTML content to the element.

The injected script is executed in the browser of the user and the DOM based XSS attack is performed and succeeds because the JavaScript code treated the user input in an unsafe way. This means that XSS vulnerabilities can be present in the website's client-side script code even with completely secure server-side code. This occurs because modern web applications (e.g., Gmail, and Facebook) become more advanced with the tendency to increasing the amount of HTML generated by script code on the client-side rather than by the server. This allows the browser to use JavaScript to refresh any part of the web page at any time without reloading the whole page, and subsequent communication with the server. A popular example is the Google Suggest: when the user start typing in Google's search box, a JavaScript sends the letters off to a server and the server returns a list of suggestions.

⁶<https://developer.mozilla.org/en-us/docs/Web/API/Document/querySelector>

⁷<https://developer.mozilla.org/en-us/docs/Web/API/Element/innerHTML>

2.3.1.4. Preventing XSS Vulnerabilities

Preventing [XSS](#) means limiting the ways in which users can affect the output of the web applications. A way to retain control of web applications is to add additional layers of protection to the web application by checking and limiting access of the web applications to external sources (e.g., `$_POST`, `$_GET`, `file_get_contents()` for PHP, [OSCs](#) and databases). Essentially, this means that a developer has to implement input validation when data enter the application and output encoding and escaping when data is send to external systems, to avoid the danger that untrusted input will have undesirable effects in application code [\[62\]](#). Data protection mechanisms can be implemented as proposed by [\[63\]](#):

- 1) **Sanitize input data:** means removing unwanted (unsafe) characters from the input data (e.g., `<`, `'`). When input data is sanitized, there are a risk of altering the data in ways that might make it unusable.
- 2) **Validate input data.** Validation confirms that the data that is coming to the web application meets the requirements of the web application (e.g., price is within expected range) and rejects data that does not meet a predefined pattern.
- 3) **Encode output data.** Output encoding involves translating special characters into some different but equivalent form that is no longer dangerous in the target interpreter. For example, translating the less than character “`<`” into the “`<`” string when writing to an [HTML](#) document.
- 4) **Escape output data.** Escaping involves adding a control characters before a character or string to avoid it being misinterpreted removing unwanted meaning, for example, adding a backslash “`\`” special character before a single quote “`'`” character so that it is interpreted as text and not as closing a string. It is common in names like “O'Brien” and “O'Reilly”.

Input validation does not always make data “safe” since certain forms of complex input may be “valid” but still dangerous. For example a valid [URL](#) may contain a [XSS](#) attack. Input validation is a technique that provides security to certain forms of data, specific to certain attacks and cannot be reliably applied as a general security rule. Additional defenses besides input validation should always be applied to data such as encoding and escaping.

The purpose of output encoding and escaping to avoid [XSS](#) is to convert untrusted input into a safe form where the input is displayed as data to the user without executing as code in the browser. This is a crucial security programming technique needed to stop [XSS](#). This defense is performed on output, when the application is building a user interface, at the last moment before untrusted data is dynamically added to [HTML](#). If this defense is performed too early in the processing of a request then the encoding or escaping may interfere with the use of the content in other parts of the program. For example, if is performed [HTML](#) escape content before storing that data in the database and the User Interface ([UI](#)) automatically escapes that data a second time then the content will not display properly due to being double escaped.

Securing [XSS](#) is a complex task, because the input coming from the outside may be used in many different contexts that also need different ways to clean the data (sanitization a variable used in a [HTML](#) tag is different than inside Javascript, for example). Moreover different contexts may be nested in several levels, increasing the overall complexity. The snippets of [HTML](#) in Table

2.2 demonstrate several examples in a variety of different contexts that need to be considered when using sanitization. For each context, a specific sanitization is required. The [OWASP XSS Prevention Cheat Sheet \[64\]](#) provides a simple positive model for preventing XSS using output escaping/encoding properly.

Table 2.2.: Snippets of [HTML](#) of untrusted data in a variety of different contexts [64].

Context	Code Sample
HTML Body	<code>UNTRUSTED DATA</code>
Safe HTML Attributes	<code><input type="text" name="fname" value="UNTRUSTED DATA"></code>
GET Parameter	<code>clickme</code>
Untrusted URL in a <code>src</code> or <code>href</code> attributes	<code>clickme</code> <code><iframe src="UNTRUSTED URL" /></code>
CSS Value	<code><STYLE> body background:url("UNTRUSTED DATA")</STYLE></code>
JavaScript Variable	<code><script>var currentValue='UNTRUSTED DATA';</script></code> <code><script>someFunction('UNTRUSTED DATA');</script></code>
JavaScript Events	<code><BODY onload=UNTRUSTED DATA></code>
DOM Based XSS	<code><script>document.write("UNTRUSTED INPUT: " + document.location.hash);</script/></code>

2.3.2. SQL Injection

[SQLi](#) is possibly the most common and widespread injection type. [SQLi](#) refers to a class of code-injection attacks in which data provided by the user is directly included in an [SQL](#) query in such a way that part of the input of the user is treated as [SQL](#) code [65]. [SQLi](#) vulnerabilities occur because the user input is not properly validated or is not validate at all. For example, the injection of [SQL](#) commands into the user and password fields of a login form can allow attackers to gain access to private data held within the database. There are two types of [SQLi](#) vulnerabilities:

- 1) **First-order [SQLi](#):** vulnerabilities occur when a non-validated input from the user is directly used to construct a [SQL](#) statement and sent immediately as a legitimate [SQL](#) query statement to the database server [65].
- 2) **Second-order [SQLi](#):** vulnerabilities occur when user input data is first stored on the back-end database and then later when that data is used in a different context to build a different [SQL](#) query. Like stored XSS, second-order injections can be especially difficult to detect and prevent because the point of injection is different from the point where the attack actually manifests itself. A developer may properly escape, type-check, and filter input that comes from the user and assume it is safe. Later on, when that data is used in a different context the previously sanitized input may result in an injection attack [65]. Second-order [SQL](#) injection attacks pose a higher risk as they are used to attack a large number of victims that access that web page, and they can also be designed to remain dormant for a period of time and not run immediately.

Web applications with [SQLi](#) vulnerabilities are extremely exposed to attacks, since they are highly requested by hackers given the benefits they can obtain. Moreover, there is an increasing number of tools (e.g., [SQLMap](#)) that automates the process of detecting and exploiting [SQLi](#)

vulnerabilities. Depending on the environment, the attacker can potentially steal, modify and can even delete sensitive data from the database, bypass authentication, compromise the web server, gain complete access to the underlying databases, systems and networks [66]. Malicious SQL statements can be introduced into a vulnerable web application using many different input sources. The most common sources of data injection are [65]:

- 1) **User input:** Web forms or HTML forms are one of the main points of interaction between a user and a web application. Forms allow users to enter data, which is generally sent to a web server for processing and storage or used on the client-side to immediately update the interface in some way. User input typically is submitted to the web server via HTTP GET and/or POST requests. Web applications access the user data contained in these requests.
- 2) **Cookies:** are data that contain state information generated by the web applications and stored in files on the client machine. When a client visits the same web application, cookies can be used to restore state information of the client. A malicious client could inject attack strings in the content of cookies. If a web application uses the content of the cookie to build SQL queries, an attacker could easily submit an attack by embedding it in the cookie.
- 3) **Server variables:** are a collection of variables that contain HTTP, network headers, and environmental variables. The client has almost total control over these variables submitted to the web application.
- 4) **Database in the back-end:** malicious user input data previously stored in the database, can be used later on to launch an attack.

Halfond et al. [65] characterized seven types of SQL Injection Attack (SQLIA), based on the goal and the intent of the attacker. Those seven types are tautologies, incorrect queries, union query, piggy-backed queries, stored procedures, inference, and alternate encoding. Depending on the specific goals of the attacker, the different types of attacks are generally not performed in isolation; many of them are used together or sequentially. Attackers can first use inference or a logically incorrect attack for database fingerprinting and then a union query (or another technique) depending on the attack target.

In the following subsections, we first present an example of a SQLi vulnerability and how can be exploited. Next, we present techniques for preventing SQLi vulnerabilities.

2.3.2.1. Exploiting SQLi Vulnerabilities

SQLi vulnerabilities have become a common issue with database-driven web applications. Most of the vulnerabilities are easily detected, and easily exploited, and as such, any web application or software package with even a minimal user base is likely to be subject to an attempted attack of this kind. Next, we introduce an illustrative example based on a basic login form. Listing 2.2 shows the HTML source code of a basic login form with two inputs (login and password), used for the authentication as user within a web application. In this form the password field is not masked for explicitness reasons. When any user submits the form, the pair username/password is sent to the PHP script “login.php” specified in the action attribute of the HTML <form> tag.

Listing 2.3 shows the source code of a possible PHP for the login.php script. The second line

```

<h1>Login Form</h1>
<form name="login" action="login.php" method="post">
  <table class='login'>
    <tr>
      <td>Username:</td><td><input type="text" name="username" value=""></td>
    </tr>
    <tr>
      <td>Password:</td><td><input type="text" name="password" value=""></td>
    </tr>
    <tr><td></td><td><input type="submit" value="Login"></td></tr>
  </table>
</form>

```

Listing 2.2: HTML code of a basic login form.

makes the connection to the database and creates the connection variable referred as `$connection` at the line 6. Lines 3 and 4 assign the values of the `username` and `password` POST fields from the user inputs of the login form to the variables `$user` and `$pass`. In line 5, these variables are directly included in the SQL query to be sent to the database server in order to verify the authentication of the legitimate user. This query instructs the database to match a table row that it has already stored, where the username and password match with the username and password of the login form. The PHP function `mysqli_query` at the line 6 sends the SQL query to the database server.

Line 7 tests for database errors and line 8 outputs these errors and ending the script. It is important to note that database error messages should never be sent to the client web browser because attackers can leverage technical details in verbose error messages to adjust their queries for successful exploitation. For instance, the code of line 8 was just written for demonstration purposes. The successful of the authentication is tested at the line 10 through the PHP function `mysqli_fetch_array` that returns, in this case, an associative array of strings that corresponds to the fetched table row or the `null` value if there are no rows in result-set. In case of success, it is initiated the process of user session creation (line 12) and other necessary operations to redirect the legitimate user to the authenticated section of the web application. Otherwise, in line 15, the user is notified with a message reporting the failure of the authentication.

```

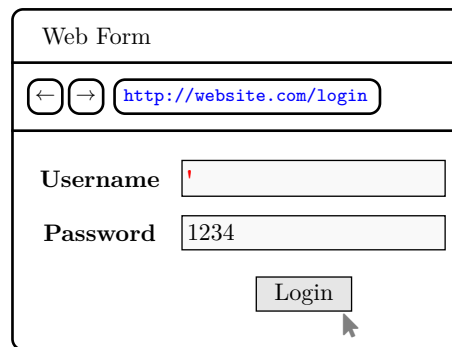
1 <?php
2 include('data_base_open.php');
3 $user = $_POST['username'];
4 $pass = $_POST['password'];
5 $sql="SELECT * FROM UserAccounts WHERE username='$user' AND password='$pass'";
6 $result = mysqli_query($connection, $sql);
7 if ($result == null ) {
8     die($connection->error);           // never send database error messages to the client web browser
9 }
10 if ($row = mysqli_fetch_array($result, MYSQLI_ASSOC)) {
11     echo "<p>Login success</p>";
12     session_start();
13     // ... register session variables, redirect, etc.
14 } else {
15     echo "<p>Invalid username/password</p>";
16 }
17 ?>

```

Listing 2.3: PHP source code for a login.php script.

To exploit [SQLi](#) vulnerabilities one needs to understand when the web application interacts with a database server in order to access some data. The attacker checks if it is possible to inject data into the web application so that it executes a user-controlled [SQL](#) query in the database. All input fields, including hidden fields, in web forms are potential injection points (i.e., [EPs](#)) whose values could be used in crafting a [SQL](#) query. For this, we need to figure out if the input fields are injection points. The very first test usually consists of adding a single quote “'” or a semicolon “;” characters to the data of the field or parameter under test. A single quote is used in [SQL](#) as a string terminator and, if not filtered by the web application, would lead to an incorrect query. The semicolon character is used to end a [SQL](#) statement and, if it is not filtered, it is also likely to generate an error.

The web form in [Figure 2.9](#) illustrates a basic test for the input field `username` by providing a single quote “'” as a username. It is very important to test each input field separately: only one parameter must vary while all the other remain constant, in order to precisely understand which parameters are vulnerable and which are not. The output errors of submitting the form using a *MariaDB* as a database server are in [Listing 2.4](#) and the output errors for *Microsoft SQL Server* as a database server are in [Listing 2.5](#).



The image shows a browser window with a web form. The address bar contains 'http://website.com/login'. The form has two input fields: 'Username' with a single quote character (') and 'Password' with '1234'. A 'Login' button is located below the fields.

Figure 2.9.: Testing for single quote.

```
SELECT username,password FROM Customers WHERE username='' AND password='1234'
```

You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '1234'' at line 1

Listing 2.4: Output error from MariaDB for enclosed quotation mark.

```
[Microsoft][ODBC Driver 17 for SQL Server][SQL Server]Incorrect syntax near '1234'.
```

```
[Microsoft][ODBC Driver 17 for SQL Server][SQL Server]Unclosed quotation mark after the character string ''.
```

Listing 2.5: Output error from SQL Server for enclosed quotation mark.

The errors above confirms that the application is indeed vulnerable to the `username` parameter. The [SQLi](#) vulnerability has been identified. At this point, it is almost certain that we will be able to extract data (it includes database names, tables names and tables data) from the back-end database of the web application. The next step is to proceed with other types of attacks that depends of the goal of the attacker, as discussed before. The following paragraphs present a illustrative example of attacks with the goal of authentication bypass.

Figure 2.10 consists of two side-by-side browser window mockups, each titled 'Web Form'. Both windows show a browser address bar with 'http://website.com/login'. Below the address bar are two input fields: 'Username' and 'Password', followed by a 'Login' button. In (a), the Username field contains 'Joe' and the Password field contains '1234'. In (b), the Username field contains 'Joe' and the Password field contains 'anything' OR '1'='1'.

(a) Authentication with a legitimate user.

(b) SQLi attack.

Figure 2.10.: Login forms filled with user data.

Figure 2.10 shows two cases of the login form filled with user data. In the left case and assuming that the pair Joe/1234 exists in the database, the result of clicking in the Login button is a legitimate user authenticating successfully. In the right case, there is an example of a SQLi attack because the values of the variables `$user` and `$pass` are gathered directly from the input of the user. The value given for the username (Joe) poses no risk, but the string provided for the password “anything' OR '1'='1” changes the conditions of the where clause of the SQL query. Therefore, as the inputs (username and password) of the login form are not properly sanitized, the use of the single quote has turned the WHERE SQL command into a three-component clause:

```

1 SELECT * FROM UserAccounts WHERE username='$user' AND password='$pass'           // SQL template
2 SELECT * FROM UserAccounts WHERE username='Joe' AND password='anything' OR '1'='1' // Concrete SQL command

```

The “OR '1'='1'” part of the WHERE clause guarantees that the WHERE condition is true regardless of what the first two parts contains. This allows the attacker to bypass the login form without actually knowing a valid username/password combination.

2.3.2.2. Preventing SQLi Vulnerabilities

Methods for preventing SQLi vulnerabilities require keeping untrusted data separate from commands and SQL queries. The user input must be sanitized before it is embedded into the queries and string concatenation should not be used to dynamically build SQL queries to reduce the possibility of code injection [41]. Techniques for preventing SQLi vulnerabilities include:

- **Use of prepared statements.** In a Database Management System (DBMS), a prepared statement is a feature used to execute the same or similar database statements repeatedly with high efficiency. Typically used with SQL statements such as queries, updates or deletes, the prepared statement (e.g., `INSERT INTO Customers (Id, Name, Email) VALUES (?, ?, ?)`) takes the form of a template with placeholders (represented by a question mark “?”) instead of concrete values for the relevant parameters. During the execution the placeholders are replaced by concrete values taking into account the data-types in the database. Any kind of data are always treated as a parameter value in the SQL query, because prepared statements have a static structure, which prevents SQL injection attacks from changing the logical structure of a prepared statement [67]. However, incorrect use of

prepared statements, like using string concatenation for constructing the template, from user input will not prevent [SQLi](#).

- **Use of stored procedures.** Stored procedures [68] are a set of instructions, written in a specific programming language, that are stored in the database itself. They can be called from the web application using its name and a list of parameters. Since Stored procedures are not always the perfect solution nor do they satisfy all the needs of all developers, other solutions exist that attempt to provide most of what a developer wants to do when accessing a database back-end. These include Object Relational Mapper ([ORM](#)) that provides an abstraction to the database without having to write data access [SQL](#) statements.
- **Whitelist input validation.** Whitelist validation is the practice of only accepting input that is known to be good. This can involve validating compliance with the expected type, length or size, numeric range, data content or other format standards before accepting the input for further processing.
- **Escaping all user supplied input.** Escaping involves adding a special character before the character/string to avoid it being misinterpreted, for example, adding a \ (backslash) character before a " (double quote) character so that it is interpreted as text and not as closing a string.

2.4. Static Code Analysis for Vulnerability Detection

Many vulnerabilities can only be detected by looking at the code. For example, executing a [SA](#) might help identify dead code, or a storage or resource leak that would be impossible to find by executing a [DA](#), or an unsafe practice (like in Java using “==” to test strings). Static code analysis is a software verification technique of examination of the source code without executing it in order to check for defects early in the [SDLC](#), avoiding costly later fixations [69]. These defects may be flaws that prevent fulfilling the software specification, but they may also be related to security problems. Static code analysis has two main approaches: manual and automated. Manual approaches involve human subjects performing the process of reviewing the code to find defects, while the automated approach uses computer-based tools in the process [70].

Manual code review is a process by which an expert is looking at the program code “line-by-line” to identify vulnerabilities. To conduct an efficient manual code review, reviewers must know all possible defects and inaccuracies before carefully checking the source code. For instance, code review requires expertise in three areas: the application architecture, the implementation techniques (programming languages, frameworks used to build the application), as well as security. Manual approaches are conducted in both formal and informal manners. The formal reviews follow a formal process that is well defined, structured and regulated. The informal reviews refer to examine software to detect defects without a prescribed process. These terms are defined more precisely in the “IEEE Standard for Software Reviews: IEEE Std 1028-2008 15 August 2008” [71]. The terms most relevant in the context of this thesis are defined as follow:

- 1) **Inspection:** is a well-defined and structured process in which a team of experts inspect a software using a systematic reading technique, in order to detect defects and identify

anomalies, including errors and deviations from standards and specifications. Determination of remedial or investigative action for an anomaly is mandatory. The inspection, is usually done by a third party of evaluators.

- 2) **Walkthroughs:** is a formal process in which a developer leads members of the development team through a segment of the code, and let the participants raise possible defects in the code. The participants ask questions, make comments, find anomalies, improve the code, consider alternative implementations, and evaluate conformance to standards or specifications. The walkthrough focuses on the presentation to an audience of the code in question by its programmer.
- 3) **Informal review or peer review:** is a process in which software are examined by any team member without any prescribed process. They are also referred to as peer reviews. The informal review is when the programmer presents his code to a colleague to review.
- 4) **Static Application Security Testing (SAST) Tools:** are computer programs used for reviewing the source code of software to identify security vulnerabilities. There are many open-source (e.g., RIPS [72], WAP [73], phpSAFE [74]) and commercial tools (e.g., Veracode White Box Testing [75], CxSAST [76], HP Fortify Static Code Analyzer [77], Sonarqube [78]) available in the market .

Code review is used to improve the maintenance process of a software product, increasing its reliability and security. Reports suggest that the cost of maintenance of software is high. For example, a study estimating software maintenance costs, found that the cost of maintenance is as high as 67% of the cost of entire SDLC [79]. The code review activity can reduce the number of required updates of the source code, and can be easily integrated into the SDLC. However, a manual review of the source code takes a very long time to do [62]. It is, therefore, recommended the use of SAST tools as they drastically reduce the time needed for code review. These tools are faster than a manual review and they are considered by many as the most efficient way to automatically locate vulnerabilities in software developed for the web [50] [80] [81] [82]. However, these tools cannot completely replace the manual review, because they have some limitations regarding the precision of the results that typically report many FPs. Some of these limitations are inherent of the static code analysis that suffers from the conceptual limitation of undecidability [83]. In fact, during static analysis, it is rather impossible to calculate the values of many dynamic string arguments (e.g., file name to an include construct) that are just defined at runtime.

The rest of the section is organized as follows. Section 2.4.1 details the concepts of Control-Flow Graph (CFG) and Data-Flow Graph (DFG), including examples. Section 2.4.2 presents static analysis techniques and related work. Section 2.4.3 presents the static taint analysis model. Section 2.4.4 presents related work on combining the results of several SAST tools.

2.4.1. Control-flow and Data-flow Graphs

Any thoroughly static analysis of the expression and data relationships in a program requires the knowledge of the CFG of the program. A CFG is a representation of all the paths that might be traversed through a program during its execution [84]. The CFG is an essential structure in a

compiler to produce optimized programs and in [SAST](#) tools to perform the data-flow analysis.

A [CFG](#) is built on top of an Abstract Syntax Tree ([AST](#)) or a Parse Tree ([PT](#)) that models the different paths of a program. In a [CFG](#), each node (or vertex) in the graph represents a basic block and each directed edge (or link) is used to represent jumps in the control flow. A basic block is a straight-line piece of code without any jumps or jump targets. In [CFG](#) it is common the use of two specially designated blocks: the *entry block*, through which control enters into the flow graph, and the *exit block*, through which all control flow leaves [51].

In the [PHP](#) example of the Listing 2.6, the variables `$_GET['product']`, `$_GET['quantity']`, `$_GET['price']` and `$_GET['bonus']` are [EPs](#). The value of the `$_GET['product']` is passed to the `$product` variable in line 2, and it is used during an `echo` call (Sensitive Sink ([SS](#))) in line 12, outputting data without any validation or encoding. In this case, the manipulation of the product variable leads to a [XSS](#) attack.

```
1 <?php
2   $product = $_GET['product'];
3   $quantity = $_GET['quantity'];
4   $price = $_GET['price'];
5   $bonus = 0;
6   if ($quantity >= 10) {
7       $bonus = intval($quantity / 10.0);
8   } else {
9       $bonus = $_GET['bonus'];
10  }
11  $total = ($quantity - $bonus) * $price ;
12  echo "Product: " . $product;
13  echo "Total to pay: $total";
14 ?>
```

Listing 2.6: PHP code example, total to pay for the purchase of products.

Figure 2.11a illustrates the [DFG](#) of the vulnerable `$product` variable, gathered from the untrusted source `$_GET['product']`, concatenated with the string “Product: ” and outputted in the line 12 without any validation or escaping. Figure 2.11b depicts the [CFG](#) of the PHP example presented in the Listing 2.11a, highlighting two execution paths. The [CFG](#) begins with an entry block (1) followed by a basic block (2). Then, the graph splits into two paths through the basic blocks 3 and 4. Next, these two paths join at the basic block 5. Finally, this block targets the End block (6).

Figure 2.12 shows two [DFGs](#) for the outputted `$total` variable that correspond to two possible execution paths for the example of Listing 2.6. In the left data flow, the value of the variable `$bonus` is calculate using the value of the variable `$quantity`. In the right data flow the value of this variable is gathered direct from the variable `$_GET['bonus']`. This is an example of path sensitive analysis combined with data flow sensitive analysis. Although no variable has been sanitized, it poses no risk of [XSS](#) vulnerabilities, but the program at runtime can report errors due to arithmetic operations with data that are not numbers.

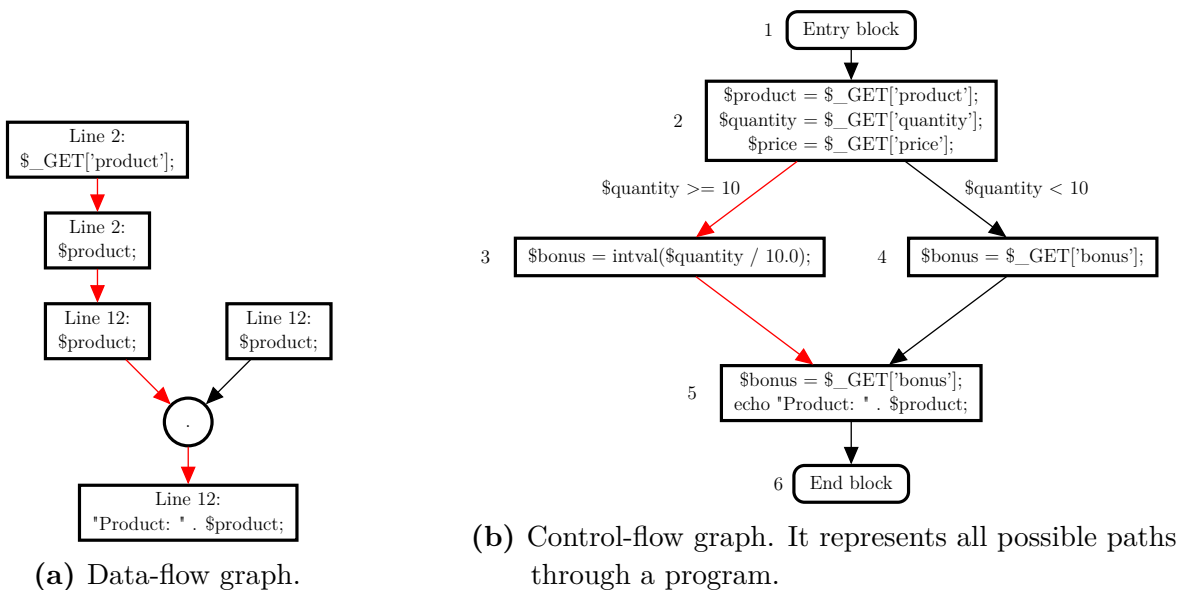


Figure 2.11.: Data-flow graph and control-flow graph for the script in listing 2.6

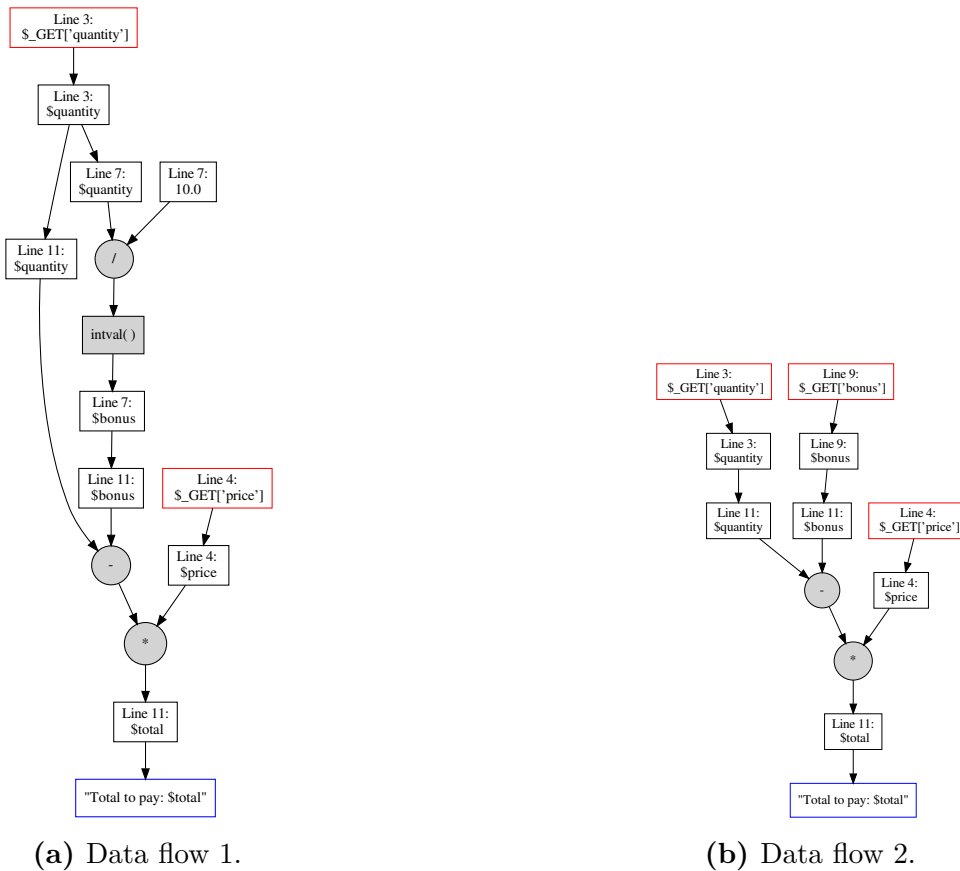


Figure 2.12.: Two data flows (execution paths) for the variable \$total.

2.4.2. Static Analysis Techniques

Performing static code analysis requires building and analyzing the **CFG** of the execution of the program. This is achieved by applying source code analysis techniques like:

- **Flow sensitive analysis:** may take into account the order of program statements and remembers what is followed by what. A flow-sensitive analysis uses the **CFG** of the source code to provide the relationship between data definition and their use. It analyzes those data, which are used after being defined. Flow sensitive analysis is more time consuming than flow insensitive analysis, but provides more precise results (i.e., less **FP** warnings) [47].
- **Path sensitive analysis:** takes into account all possible paths of execution of the program by considering all conditional branch instructions [85]. Path sensitivity analysis can be very expensive (exponential number of paths) but provide higher precision.
- **Context sensitive analysis:** commonly divided into two techniques [47]:
 - **Inter-procedural or global analysis.** It analysis functions by considering global program variables and actual parameter of the function call and models the relationships between various functions. Tainted data can reach a function from its parameters, user input variables, other functions, and from global variables, so it depends on the global state of the program. The analysis also verifies if the function is able to sanitize the tainted data.
 - **Intra-procedural or local analysis.** This follows the same procedure as the inter-procedural analysis, but it only processes the inside of the function, without considering the context from which the function is called. Intra-procedural analysis algorithm only models information flow that does not cross function boundaries, so results in too much false positives and false negatives. An intra-procedural analysis is faster than inter-procedural analysis, but provide less precision than inter-procedural analysis.
- **Functions summaries.** A function is parsed only once. The summary of this analysis is reused in subsequent calls to determine the effects on the context of the calling code.
- **Whole-program analysis.** A function is parsed every time it is called. A way to perform this is to replace each function call by the function body (inline function), which results in a huge program. Consequently, this method requires a lot of memory and processing power.
- **Static string sensitive analysis.** The main functionality of a web application is often string processing [86]. The inputs from the client-side are mostly strings. The outputs of web applications are mainly strings as well, such as **SQL** queries, **HTML** pages, and **JavaScript** code pieces. String analysis is a static analysis technique that determines the values that a string expression can take during program execution at a given program location. String analysis involves tracking all of the possible values that strings may contain while they traverse through the code [87]. String analysis could lead to less false positives and less false negatives.
- **Field (or instance) sensitive analysis.** An analysis is field sensitive, if different members of the instantiated objects are considered as separate variables [88]. For example, an object data type (e.g., instance of a class in **PHP** language) might have two members of string variables. One of the variables contains untrusted data while the other contains trusted data.

Without field sensitivity, the entire instance of the class would be considered untrusted (or tainted) as member variables are not distinguished. Field sensitivity allows reducing false positive warnings [89].

- **Points-to-analysis (or pointer analysis, or alias analysis)**. Whenever a variable is assigned a tainted value, this taint value is propagated to all its aliases (variables pointing to the same memory location, which is a pointer in C language) [81]. This includes variables and objects passed to functions and methods by reference and functions and methods that return values (variables or objects) by reference.

Researchers have been using different techniques in their works. Christensen et al. applied string analysis to detect [SQLi](#) vulnerabilities by analyzing the source code in Java programs [90]. Minamide extended these techniques and presented a string analysis technique that approximates the string output ([HTML](#)) of a PHP web application with a context-free grammar for detecting [XSS](#) vulnerabilities [91]. Huang et al., the pioneers of static analysis, developed a tool called [WebSSARI](#) [80]. It employs flow-sensitive, intra-procedural analysis based on a lattice model of security levels to track the taintedness of variables through the program. [WebSSARI](#) has been used to find a number of security vulnerabilities in PHP scripts, but they used only an intra-procedural taint analysis algorithm and thus only models information flow that does not cross function boundaries, so results have too much [FPs](#) and [FNs](#). It was made unavailable in 2006 and further advances were implemented in the commercial tool [CodeSecure](#) [92].

Livshits and Lam proposed a [SA](#) approach based on precise context-sensitive (but flow-insensitive) and points-to analysis for analyzing the bytecode of Java web applications [93]. They used a high-level declarative language Program Query Language ([PQL](#)) to specify vulnerabilities (through information flow policies) that were automatically translated into static analyzers. These static analyzers were automatically used for detecting [SQLi](#), [XSS](#) and [HTTP](#) splitting vulnerabilities. The results of [SA](#) are presented to the user for assessment in an auditing interface integrated within the Eclipse [IDE](#). The experimental results of the analysis of a benchmark composed by nine large, popular open-source applications, showed that the proposed analysis is an effective practical tool for finding security vulnerabilities. The tool reported [FPs](#) only in one of the nine benchmarks used. One limitation of their static analysis approach is that it does not model control flow in the program and therefore may miss-flag sanitized input that subsequently flows into [SQL](#) queries.

Xie and Aiken [94] presented an inter-procedural and flow-sensitive [SA](#) algorithm for PHP. The algorithm performs a bottom-up analysis of basic blocks, procedures (intra-procedural), and the whole program to find [SQLi](#) vulnerabilities. It is also capable to handle with dynamic features unique to scripting languages such as dynamic typing and code inclusion. Their technique employs symbolic execution⁸ to automatically derive the set of variables that need to be sanitized before utilized by functions. The results demonstrate the effectiveness of their approach on six popular open source PHP code bases, finding 105 previously unknown security vulnerabilities.

Jovanovic et al. developed [Pixy](#), a Java tool to detect [SQLi](#) and [XSS](#) vulnerabilities [81]. [Pixy](#)

⁸Symbolic execution consists in executing the program while keeping input variables symbolic rather than assigning them a value. In this way, one can derive path invariants, i.e. properties on the variables that are always true for a given path.

uses a flow-sensitive, inter-procedural and context-sensitive data flow analysis to determine if user data reach **SSs** sensitive sinks without being fully sanitized. It performs precise alias and literal analysis to refine the taint process and improve the precision of the detection, but it does not parse object oriented constructs and PHP dynamic features (e.g., file including automatically). **Pixy** is a command line tool and provides a text-based report of the vulnerabilities offering several verbosity levels. It also reports the **CFG** of the analyzed code and the **DFG** of each vulnerability detected in a plain text graph description language. The results show a **FP** rate of around 50%.

Balzarotti et al. [83] extended **Pixy** to perform analysis of the sanitization process and thus are able to deal with a custom sanitization. More precisely, they combine **SA** and **DA** techniques to identify faulty sanitization procedures that can be bypassed by an attacker. They perform string analysis through language-based replacement and represent values of variables at concrete program locations using finite state automata. The approach was implemented in a tool, called **Saner**, and applied to a number of real-world applications. The results demonstrate that the tool was able to identify several novel vulnerabilities that stem from erroneous sanitization procedures. In fact, whenever a web application applies some sanitization routine to potentially malicious input, some of the vulnerability analysis assume that the result is innocuous. Unfortunately, this might not be the case, as the sanitization process itself could be incorrect or incomplete. The **SA** that they used was based on **Pixy**, therefore it has similar limitations as **Pixy**. Moreover, the database may not contain strings corresponding to attacks that were not considered in advance. Consequently, it can both miss vulnerabilities and cause false positives.

Yu et al. [95] developed an automata-based approach for finding and eliminating string-related security vulnerabilities (**SQLi**, **XSS** and Malicious File Execution (**MFE**)) in PHP applications. It incorporates the widening operator⁹ to tackle the problem of handling variables updated in loops. The approach was implemented in a tool, called **STRANGER** (STRing AutomatoN GEnerator). The tool uses **Pixy** as a front end and **MONA**¹⁰ automata package for finit-state automata manipulation. **STRANGER** implements an automata-based approach [96][97] for automatic verification of string manipulating programs based on symbolic string analysis. **STRANGER** uses symbolic forward and backward reachability analyses to compute the possible values that the string expressions can take during program execution. The authors experimented the tool on several small benchmarks extracted from known vulnerable web applications [97]. For each vulnerable benchmark, the authors also generated a modified version where string manipulation errors are fixed. **STRANGER** took less than a few seconds to analyze each benchmark. Unfortunately, when analyzing a real web application (PHP guestbook, SimpGB-1.49.0) composed by 153 PHP files containing over 44,000 lines of code the tool took several hours performing the analysis.

Wassermann et al. proposed an approach to detect **SQLi** and **XSS** vulnerabilities by combining tainted information flow with string analysis [82]. Their approach labels and tracks untrusted substrings from user inputs, ensuring that no untrusted scripts can be included in **SQL** queries and generated **HTML** pages. Moreover, their approach checks web applications against a policy that no untrusted data should invoke the **JavaScript** interpreter, represented by a black-list.

⁹A widening operator's purpose is to compress infinite chains (e.g., values of a valuable in a loop) to a chain with finite length. Widening operators play a crucial role in particular when infinite abstract domains are considered to ensure the scalability of the analysis to large software systems.

¹⁰<https://www.brics.dk/mona>

Xin-Hua Zhang et al. implemented a taint analysis based tool (ASPWC) to detect [SQLi](#) and [XSS](#) vulnerabilities in ASP.NET source code [98]. It tracks various kinds of external inputs, tags taint types, constructs [CFGs](#) based on the use of data flow analysis of the relevant information and taint data propagation to various kinds of vulnerability functions. The results showed that the detection approach is effective to detect [SQLi](#) and [XSS](#) vulnerabilities, but with a high false positive rate.

David Hauzar and Jan Kofron [99] proposed a method for the identification of bugs inside web applications caused by data flow of unsanitized inputs from the user to [SS](#) inside web applications written in PHP. The authors argued that the state-of-the-art tools for bug discovery in languages used for web-application development, such as PHP, suffer from a relatively high [FP](#) rate and low coverage of real errors. They refer that this is caused mainly by unprecise modeling of the tools of dynamic features of such languages and path-insensitivity. The approach proposed by the authors combined known techniques such as precise modeling of aliasing and taint analysis. Additionally, they performed path-sensitivity analysis to reduce [FP](#) caused by path-insensitivity analysis of existing tools (e.g., [Pixy](#)). The approach is based on analysis that computes data flow information using dependence graphs, identifies [EP](#), [SSs](#), and at each program location maintains: the taint and the sanitization status for each variable, the set of possible values of each variable, the set of conditions defined on the variables of the program that must hold, and the set of possible types of each variable. During the analysis are collected the conditions that must hold in order for these program locations to lead to a vulnerability and the conditions that must hold to reach the critical command corresponding to the vulnerability. Then it was used an Satisfiability Modulo Theories ([SMT](#)) solver to find a solution of the conjunction of these conditions. If the [SMT](#) solver proves these conditions unsatisfiable, the vulnerability is unfeasible. However, if the [SMT](#) solver proves the formula, the vulnerability can be still unfeasible resulting, due to the dependencies between variables in the conditions. The approach was not evaluated in real web applications and their path-sensitive analysis is very expensive and may produce [FPs](#).

In another study, David Hauzar and Jan Kofron [100] proposed a [SA](#) framework for PHP based on abstract interpretation, automatically resolving features common to dynamic languages (e.g., dynamic type system, indirect variable use, virtual and dynamic method calls, dynamic includes, dynamic object and function declaration) and thus reducing the complexity of defining new static analyses. The framework automatically resolves dynamic features and makes it possible to define static analyses without taking these features explicitly into account. The framework was evaluated using two web applications and the results compared with [Pixy](#) [81] and [Phantom](#) [101]. The results showed that the framework outperformed the other tools both in vulnerability detection and number of [FPs](#).

Y. Zheng and X. Zhang proposed a path-sensitive and context-sensitive inter-procedural analysis to detect Remote Code Execution ([RCE](#)) vulnerabilities [86]. The analysis features a novel way of reasoning both the string and non-string behavior of a web application in a path sensitive fashion. The developed prototype system evaluated on ten real-world PHP web applications reported 21 true [RCE](#) vulnerabilities, with eight previously unknown.

Dahse and Holz developed [RIPS](#) [102], a tool based on the specificities of the PHP language that performs a comprehensive analysis and simulation of built-in language features, such as PHP functions, taking into account only the called arguments that have to be traced [51]. It also

includes information about user input variables, **SSs**, sanitization functions, secure an unsecure PHP built-in functions, and other PHP features. Furthermore, **RIPS** performs a context-sensitive string analysis based on the current markup context, source type, and PHP configuration. It is based on the **AST** of the PHP script and performs intra-procedural and inter-procedural analysis to create the respective **CFG**. **RIPS** is able to perform backward-directed taint analysis for 20 different classes of vulnerabilities, including **XSS** and **SQLi**. However, the tool does not parse PHP objects and, consequently, it misses encapsulated vulnerabilities in modern **OOP** based web applications and plugins. **RIPS** has only been developed as open source until 2014, and in 2016 it was released a commercial version able to fully analyze **OOP** code.

Medeiros et al. [103] proposed the tool **WAP** using a hybrid of methods to both detect and fix input validation vulnerabilities in **OOP** PHP source code such **XSS** (first and second orders), **SQLi**, Local File Inclusion (**LFI**) and Remote File Inclusion (**RFI**), Parse Tree (**PT**) and OS Command Injection (**OSCI**). The first step of their approach uses taint analysis to flag candidate vulnerabilities based on configurable **EPs**, **SSs**, and sanitization functions. The second step uses data mining (classification) to refine the results obtained, therefore reducing the number of false positives. This process is based on a dataset with 76 vulnerabilities with 15 attributes. The precision of the tool directly depends of the dataset and can be improve whenever a vulnerability or **FP** is verified to be so. Next, it identifies the right places for correcting the source code and finally the vulnerabilities are automatically removed using code fixes essentially based on proper validation or sanitization of user input. The tool was evaluated with a large set of 35 open source PHP applications with more than 2,800 files and 470,000 lines of code and found 294 vulnerabilities (at least 28 of which were **FPS**) in 107 files. The results shown that **WAP** was able to process large PHP applications and corrected all the vulnerabilities it detected. The results showed that the second stage (data mining) of the **WAP**'s analysis improved their accuracy from 69% to 92.1%. The results of comparing **WAP** with two well-known tools in the literature, **Pixy** [81] and **PhpMinerII** [104], revealed that **WAP**'s accuracy and precision were approximately 5% better than **PhpMinerII**'s and 45% better than **Pixy**'s.

Table 2.3 shows a summary of approaches covered in this section, depicting the techniques used for each approach, the programming languages and classes of vulnerabilities. We observed that these approaches used diverse techniques as a way to increase the rate of vulnerabilities reported and to reduce the rate of **FPS**.

2.4.3. Static Taint Analysis

The most common model for detecting input validation vulnerabilities in programs is the *taint analysis* model. This model, also called *taint checking*, was implemented both by means of static [105], [81], [93], [80], [51] and dynamic analysis [106] [107]. Taint analysis is a special case of data flow analysis that allows tracking unverified external data (e.g., user input) distribution across the program. If such data, without any validation, gets into the code key points (e.g., function manipulating data in a database) it may lead to various vulnerabilities, including **SQLi**, **XSS**, path traversal and others [47]. To perform Static Taint Analysis, two concepts play an important role:

Table 2.3.: Summary of existing static analysis techniques.

Authors	Tool	Year	Analysis Techniques								Lang uage	Vulnerabilities			
			A	B	C	D	E	F	G	H		SQLi	XSS	Others	
Christensen et al. [90]	-	2003									×	Java	×		
Huang et al. [80].	WebSSARI	2004			×							PHP		×	
Minamide [91]	-	2005									×	PHP		×	
Livshits et al. [93].	-	2005		×				×				Java	×	×	×
Yichen Xie et al. [94]	-	2006	×	×	×							PHP		×	
Jovanovic et al. [81]	Pixy	2006	×	×				×	×			PHP		×	
Balzarotti et al. [83]	Saner	2008	×	×				×	×			PHP		×	
David et al. [99]		2012	×	×				×	×			PHP	x	×	x
Yu et al. [95]	Stranger	2008	×	×				×	×			PHP	-	×	-
Wassermann et al. [82]	-	2008								×		PHP		×	
Xin-Hua et al. [98]	ASPWC	2010		×								ASP	×	×	
Zheng et al. [86]	-	2013	×	×		×				×		PHP	-	×	×
Dahse and Holz [102]	RIPS	2014	×	×	×	×				×		PHP	×	×	×
Medeiros et al. [103]	WAP	2014	×	×	×	×				×		PHP	×	×	×
David et al. [100]	Weverca	2015	×	×				×	×			PHP	x	×	x

A - Flow Sensitive Analysis

B - Inter-procedural Analysis

C - Intra-procedural Analysis

D - Path Sensitive Analysis

E - Pointer Analysis

F - Alias Analysis

G - Literal Analysis

H - String Analysis

- 1) **Entry Point (EP)**: is the location where the external data enters into the software. Data may come from an insecure source, such as the attacker, the network, the database, a file, or other software components. Examples of EPs in web applications are the locations of the GET and POST HTTP parameters.
- 2) **Sensitive Sink (SS)**: is the location inside a program where a vulnerable function is called. A vulnerable function is a function that exposes private data to external systems that could cause harm. When an unverified EP controlled by the attacker is passed to a vulnerable function, we have a vulnerability. An example of a SS for SQLi vulnerability is the PHP “mysql_query” function, which executes a SQL query and returns the results. The PHP “print” and “echo” functions that outputs HTML, CSS and JavaScript to the browser are examples of SSs for XSS vulnerabilities.

Next we introduce some important concepts in the topic of tainted analysis:

- 1) **Taint data**: this term refers to the values that an attacker can use for unauthorized and malicious operations when interacting with the vulnerable program.
- 2) **Taint source**: is the EP where the tainted data enters the program.
- 3) **Taint sinks**: is the SS where the tainted data is used to cause harm.

Static taint analysis uses the concept of *taint data* to locate vulnerabilities [81] [62]. Taint data analysis starts with variables that come from a taint source, which can be maliciously manipulated from the outside [62]. When a tainted variable is used by the program in a Sensitive Sink (SS) (a taint sink), an attack becomes possible.

The data flow between EPs and SSs can be modeled and analyzed using several source code

analysis techniques, such as flow sensitive analysis, path sensitive analysis, context sensitive analysis, inter-procedural and context-sensitive data flow analysis [83]. During the data flow process, tainted data may propagate to other program variables, making them also tainted. On the other side, tainted variables may become untainted using a validation process that is dependent on the specificity of the tainted variable and on the vulnerability type being prevented. However, these variables may be tainted again due to functions that reverse the validation process performed previously.

2.4.4. Combining Static Analysis Tools

The state-of-the-art of **SAST** tools are not a silver bullet and, on average, only able to detect about half of the existing security vulnerabilities [1]. In fact, **SAST** tools have limitations, such as missing some of the vulnerabilities (**FN**) and generating many **FP** [108]. It is known that different **SAST** tools report distinct sets of security vulnerabilities, with some overlap [51] [103] [109]. To improve their overall detection capabilities, some researchers have proposed combining the results of diverse **SASTs** to improve the overall detection.

Rutar et al. [110] studied five well-known **SAST** tools on a small set of Java programs with different sizes from various domains. They concluded that the results of each tool were highly correlated with the techniques used for finding bugs, and that no single tool could be considered the best to detect defects. They proposed a meta-tool based on a set of scripts for automatically combining and correlating the outputs of various tools in a common format. The vulnerabilities found were not manually reviewed, thus, there was no distinction between True Positive (**TP**) and **FP**. The metric used to evaluate and compare the tools was the number of vulnerabilities found by each tool.

Meng et al. [111] proposed an approach to merge the results of multiple **SAST** tools. The user specifies the programs to be analyzed and chooses the classes of bugs to be scanned. After determined which tools could search for the specified class of vulnerability, the necessary configurations to run the tools was generated, the tools executed, the outputs combined in a single report, and two prioritizing policies to rank the results were applied. Meng et al. concluded that developers could benefit from more than one **SAST** tool. The results were not classified as **TP** and **FP** and the authors did not propose any metric to evaluate the approach. The dataset was composed by a small Java program that was not representative of real applications. Therefore, with such limited validation it is very difficult to assess the strength and drawbacks of the solution.

Wang et al. [112] proposed an approach that combines multiple **SAST** tools in a Web Service (**WS**). The user had the possibility to choose the classes of vulnerabilities to scan and upload the source code and auxiliary information such as the programming language and the classes of vulnerabilities to be scanned. The tools perform the analysis of the source code and results are merged in a way that the same defect is only reported once. The combined results are sent back to the user. The approach was evaluated in terms of running time when combining two **SAST** tools, but the experiments were quite limited, having just a single Java test case. Therefore, the solution lacks a proper validation of its effectiveness.

The National Security Agency (NSA) Center for Assured Software (CAS) specified a methodology, the Center for Assured Software (CAS), that measures and rates the effectiveness of SAST tools in a standard and repeatable manner [113]. The main goal of the CAS is to provide objective information to organizations that want to purchase commercial SAST tools or to use free ones. The metrics used for evaluating the SAST tools are *Precision*, *Recall*, *F-Score* (i.e., *F-Measure*), and *Discrimination Rate (DR)*. A discrimination occurs if a SAST tool reports a vulnerability in the vulnerable code and keeps quiet in the non-vulnerable code, a fixed version of the vulnerable code. The CAS has created a collection over 81,000 synthetic C/C++ and Java programs with known vulnerabilities, which is called Juliet Test Suite and available online [114]. Each test case is a slice of artificial code having exactly one vulnerability and at least one non-vulnerable construct similar to the vulnerability. In 2011, the CAS conducted a study with the purpose of determining the capabilities of five SAST tools for C/C++ and Java [115]. In this study, they proposed the combination of two SAST tools to show that adding a second SAST tool might complement the first one. However, the evaluation of the combinations was limited because it was based on the recall and DR metrics. The recall metric does not consider the number of FPs reported, and the DR severely penalizes SAST tools that report many vulnerabilities but also report FPs. Furthermore, they also evaluated the overall recall of four combinations of SAST tools. The SAST tools were labeled with a number from 1 to 5. Then, the combination of SAST tools: 12, 123, 1234, and 12345, were evaluated across all the test cases. They concluded that the recall increases as the number of tools increases. However, this evaluation is limited as there are many combinations that were not considered.

Diaz et al. [116], compared the performance of nine SAST tools, most of them commercial tools, against the Software Assurance Reference Dataset (SARD) from the SAMATE [117] at NIST. Based on the results, the authors recommended the use of several SAST tools with different detection algorithms/heuristics to improve the results.

Beller et al. [118] investigated how common was the use of SASTs in real-world, taking as reference the 122 most popular Open-Source Software projects. The results showed that a single SAST tool was used in 41% of the projects, two SASTs in only 22%, and three SASTs tools in 14% of the projects. This suggests that developers might not be aware of the benefits of using multiple SAST tools and/or that the increase of FPs reported may lead developers to avoid using multiple SAST tools [119].

2.5. Dynamic and Hybrid Security Analysis

Dynamic Analysis (DA) is the analysis of the properties of a running program. It observes the behavior of the running program and takes into account the inputs and the outputs to identify security vulnerabilities in the current execution path [120]. DA approaches for vulnerability detection such as penetration testing are based on the web application analysis from the user-side. It requires the web application running and is independent of the programming language used to develop the web application. DA does not require access to its source code, byte code, or binaries. This technique sends malicious patterns (special input test cases) into the web application and analyzes the results. If any errors are observed, then an assumption of possible vulnerability is

made. For some cases, it can be difficult to properly analyze the results because a test case can have effects that are not directly visible on the outputted results. For example, an update or a delete operation in the back-end database of the web application, may only be detected by inspecting directly the database data.

DA is usually used during the deployment phase by the end of the **SDLC**, ensuring the web application runtime protection. Its usefulness derives from two of its essential characteristics [121]:

- 1) **Precision of information.** **DA** normally consists of instrumenting a program to observe or record certain aspects of its run-time state. This instrumentation can be tuned to collect precisely the information needed to address a particular problem, for example, to analyze if the user input data reach a certain output in a specific location of a program. An instrumentation tool can be created to verify the pairs of input/output data at this locations of the program.
- 2) **Dependence on program inputs.** **DA** provides a powerful mechanism to correlate inputs and outputs with program behavior [120]. With **DA** it is easy, in most cases, to map changes in program inputs with changes in the internal program behavior and program outputs, since all of them are directly observable and linked by the program execution.

Dynamic analysis has some limitations, like the detection of only the vulnerabilities present in the actual execution paths, leaving undetected vulnerabilities in parts of code that were not executed. **DA** cannot be used for the detection of indirect control dependencies, but it is precise because it generates less **FPS** [122]. Another disadvantage of the dynamic approach is the large amount of computational resources required to execute the analysis [123]. In fact, both static analysis and dynamic analysis comes with limitations and strengths and are complementary in several aspects. This is precisely where hybrid analysis methods come into play: to combine techniques in a way that leads to higher detection rates and coverage, lower false positive rates, and more efficiency [124].

The rest of this section is organized as follows. Section 2.5.1 presents taint-base protection techniques. Section 2.5.2 presents taint-free protection techniques. Section 2.5.3 presents black-box and white-box testing techniques. Section 2.5.4 introduces the concepts of hybrid analysis techniques and presents related work.

2.5.1. Taint-based Protection

Dynamic taint analysis tracks data flow in a running program. This technique marks input data from unsafe inputs as tainted, and then propagates that taint data to other values that are computed transitively based on these tainted inputs [125]. For instance, this is used for detecting vulnerabilities in software by marking **EPs** as tainted, and then checking whether they propagate to inappropriate outputs (**SSs**). Dynamic taint analysis usually requires instrumentation of the source code or other components (e.g., interpreter, library) to collect the information. As a result, the space and time overheads may negatively affect the performance and stability of the running web application.

Perl and Ruby are two major programming languages that provide built-in support for dynamic taint tracking [106] [126]. Taint mode of Perl is one of the best-known examples of dynamic taint propagation analysis to web applications. All command-line arguments, environment variables, locale information, results of certain system calls, and all file input are marked as *tainted*. Tainted data may not be used directly or indirectly in any command that invokes a sub-shell, nor in any command does that modify files, directories, or processes.

Nguyen-Tuong et al. [127] proposed a taint mode for PHP-based web applications to prevent injection attacks such as XSS and SQLi. They modified the PHP interpreter's implementation of the string data type to include tainting information for string values at the granularity of individual characters. It tracks each character in the user input individually, and employs a set of heuristics to determine whether a query is safe when it contains fragments of user input. For example, it detects an SQLi vulnerability if an operator symbol (e.g., “(”, “)”, “%”, “;”, “'”) is marked as tainted. To prevent XSS vulnerabilities they modify the PHP output functions (`print`, `echo`, `printf` and other printing functions) with functions that check for tainted output containing dangerous content. However, this approach is still susceptible to both FPs and FNs. Initial measurements indicate that the performance overhead incurred by using the modified interpreter is less than 10%.

Chess et al. [128] proposed a dynamic taint-based approach to detect input validation vulnerabilities such as SQLi and XSS. The target program is monitored in order to track untrusted user input. Then, this untrusted input is inspected to check its validity before the use during the execution of the program. The results showed that their approach achieves higher test coverage (and therefore finds more vulnerabilities) than typical security testing techniques.

Haldar et al. [107], proposed a dynamic tainted analysis that tags and tracks user input at runtime and prevents its improper use to maliciously affect the execution of the program. It treats all user input as tainted data, and it detects malicious code when the data do not match up with the registered patterns. Benchmarks using Java programs show that the overhead of this runtime enforcement is negligible and can prevent a number of attacks.

2.5.2. Tainted-free Protection

Halfond et al. [129] proposed a technique which uses a model-based approach (AMNESIA) to detect illegal queries before they get executed on the database. AMNESIA, first extracts from the PHP source code the structure of legitimate SQL queries to build a model for each vulnerability. Then, using runtime monitoring it compares the dynamically generated queries with the model and, if they match, the monitor lets the query be executed, otherwise the execution of the query is not permitted. This approach gave no FPs and detected 1,470 attacks performed by 3,500 legitimate accesses to the applications analyzed.

Bandhakavi et al. [130] proposed a mechanism, called CANDID, which retrofits web applications written in Java through a program transformation to defend them against SQLi attacks. It detects command injections using shadow query strings¹¹ instead of tracking taint information directly.

¹¹In a shadow string, all characters c originating from a program are remapped to shadow characters sc , where

CANDID employs [DA](#) to extract the structure of [SQL](#) queries by feeding benign candidate inputs into the web application. It consists of two components: an offline [Java](#) program transformer that is used to instrument the web application, and an online [SQL](#) parse tree checker. Thus, the web application is instrumented at each query generation location with a shadow query, which captures the intended programmer structure. The user input within the shadow query strings is replaced with known non-attack strings such as a sequence of the character “a”. Then, at runtime, the structure of the generated [SQL](#) queries are also captured and the [SQL](#) checker is invoked to compare the structure of the queries. Any structural difference in the parse tree of the real and shadow queries reveals an attack.

2.5.3. Black-box and White-box Testing

Black-box testing is a [DA](#) technique that make use of both benign and malicious test inputs to try to construct input vectors that expose input validation vulnerabilities. In black-box testing, the source code is not examined. Instead, special input test cases are generated and sent to the application. Then, the results returned by the application are analyzed for unexpected behavior that indicates errors or vulnerabilities [\[131\]](#). For example, these inputs are provided to the web application and it is observed if the outputs are successfully tampered with the malicious inputs.

Black-box testing techniques are a common approach to improve software quality and detect bugs before deployment. It has the potential of finding most types of defects, however, testing is costly and is likely to leave defects undetected [\[132\]](#). The generation of the test inputs to cover all the code of the web application is a challenging task. Without the knowledge of the internals of the web application it is very difficult to uncover subtle vulnerabilities such as defective sanitization.

Doupé et al. [\[133\]](#) compared eleven security scanners. The results of the evaluation show that crawling is a critical task and compromises the ability of the scanners to detect vulnerabilities. Thus, more sophisticated algorithms are needed to perform “deep” crawling and track the state of the technologies used to develop the application under test. They concluded that black-box scanners struggle to crawl the applications deep enough to identify various vulnerabilities and that providing meaningful/valid test data is challenging. The rate of [FNs](#) is typically between 60% and 90% [\[134\]](#).

White-box testing is one of the biggest techniques used today. It is typically very effective in validating design and finding programming errors and implementation errors in software. White-box testing is performed based on the knowledge of how the web applications are implemented. It makes use of functional specifications, detailed designing of documents, source code, and security specifications to generate test cases to exercise the internal of the program [\[135\]](#). Therefore, it requires a deep level of understanding of the source code and of the application, and several techniques to derive test cases in order to test every visible execution path and determine the appropriate outputs. Thus, white-box testing is very computational expensive and, on some occasions, it is not realistic to test every single existing condition of the web application and

sc = $map(c)$, while all characters originating from user input remain intact. Value shadowing is a precise, lightweight way to propagate character level taint information.

some conditions will be untested.

Kiezun et al. [136] proposed an automatic technique for creating inputs that expose [SQLi](#) and [XSS](#) (first-order and second-order) vulnerabilities and a tool ([Ardilla](#)) that implements the technique for PHP/MySQL web applications. The technique requires the source code of the application and it is based on input generation, dynamic taint propagation, and input mutation to find a variant of the input that exposes a vulnerability. [Ardilla](#) creates concrete attack vectors by systematically mutate inputs that propagate taints to [SSs](#), using a library of strings that can induce [SQLi](#) and [XSS](#) attacks. After, it checks if every flow of tainted data to a [SS](#) propagates the untrusted data indicating a vulnerability or if the data are sanitized through the routines in place. The evaluation of the tool on five PHP applications found 68 previously unknown vulnerabilities (23 [SQLi](#), 33 first-order [XSS](#), and 12 second-order [XSS](#)).

Ciampa et al. [137] proposed an approach and a tool for web application penetration testing. The user provides the base [URL](#) of the target web application and the tool automatically crawls the application and downloads the resulting pages. Then, it identifies the input parameters defined within [HTML](#) forms, which are the potential injection points. For each one, the tool sends a series of generic injection strings and, based on the response of the application determines if it is vulnerable. In case of success, it injects more specific attack strings to infer the [DBMS](#) and the structure of the database. The tool was evaluated using 12 real web applications. Although the results were better than the well-known [SQLMap](#) tool, the study has some limitations, like not taking into consideration the [FPs](#).

2.5.4. Hybrid Analysis

No single automated analysis technique (tool) can find all possible vulnerabilities: each technique has its own strengths and weaknesses [129] [138] [139]. [SA](#) and [DA](#) come with their advantages and disadvantages and they can complement each other, since each one is able to detect to a greater or lesser extent each class of vulnerability for which they are designed for. In fact, each technique detects different sets of vulnerabilities with some overlap. The diagram of the figure 2.13 illustrates the classes of vulnerabilities each technique can discover, and where they overlap for correlation purposes. The large green circle indicates the total potential security vulnerabilities that could be found in the software. The small pink circle means that [SA](#) and [DA](#) together may fail detecting some vulnerabilities that can be manually detected by experts.

[HA](#) refers to the combination of [SA](#) results (which identify potential vulnerabilities) with [DA](#) results (which identify which threats are actually exploitable). In this context, the [SA](#) could identify the vulnerabilities candidates and the [DA](#) could then be used to limit the amount of candidates [123]. For the most comprehensive code coverage and vulnerability coverage, multiple [SA](#) and [DA](#) techniques should be used. This combination is an approach that many penetration testers are leveraging today [141]. In fact, [SA](#) and [DA](#) are complementary techniques in a number of dimensions:

- **Completeness.** [SA](#) techniques can provide 100 percent code coverage (unlike [DA](#)), but they do not provide 100 percent vulnerability coverage [123]. Since [DA](#) examines actual path program executions, it does not suffer from the problem of infeasible paths that can

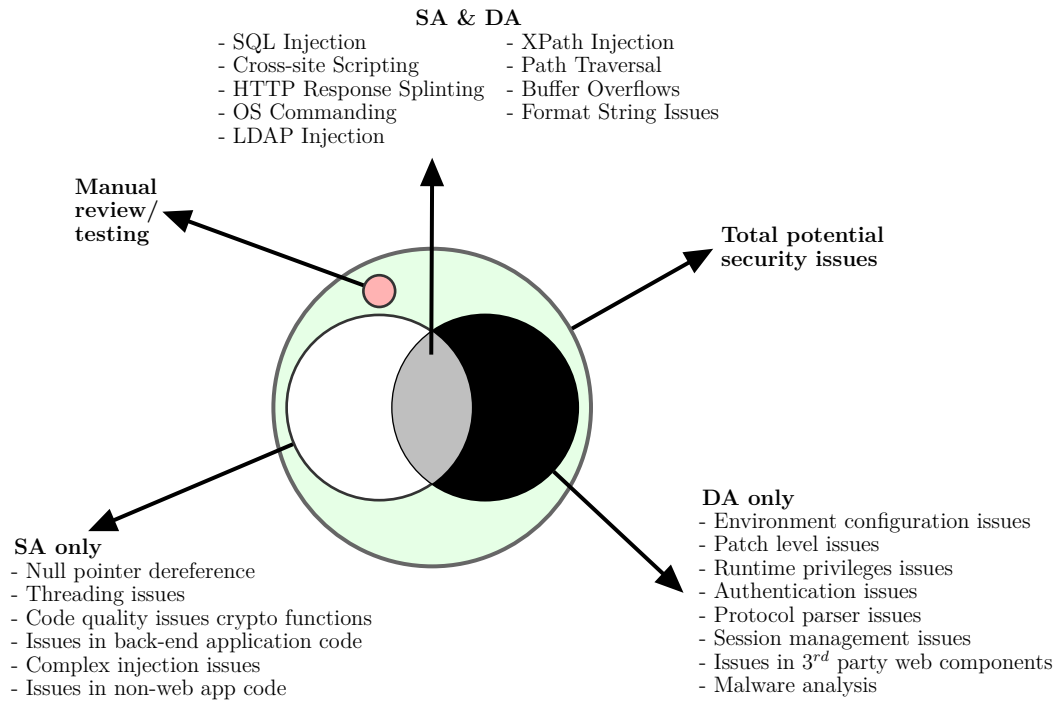


Figure 2.13.: SA and DA: Issue type coverage. Adapted from [140].

plague SA reporting many FPs.

- **Scope.** DA can examine very long program paths, so it has the potential to discover semantic dependencies between program entities widely separated in the path, including those outside the code and in third-party interfaces. SA is typically restricted in the scope of a program it can analyze effectively and efficiently. It may be difficult to discover such “dependencies at a distance” due simplifications of the SA models.
- **Precision.** DA is precise because it examines the concrete domain of the program execution allowing to collect precise information about the behavior of the program, but it faces a hard time in finding paths that activate the vulnerability [142]. SA observes mainly the program structure. Thus, it can show if a given method could be called, but it cannot provide information on how often or even if it will be called at runtime. Modern web applications are developed using object-oriented programming, a powerful methodology to manage complex systems, which is known to be harder to analyze than imperative languages due to the use of classes and objects [103]. DA is more precise, specially in handling object-oriented features like encapsulation, inheritance and polymorphism, but it is, at the same time, a slow and complex process in comparison to SA [143].

Static and dynamic analysis have complementary advantages, and this has led researchers to devise combinations to achieve the best of both worlds (e.g., [138], [83], [137], [139]). For example, Su and Wassermann [138] proposed an algorithm (SQLCheck) for preventing SQLi attacks based on context-free grammars and compiler parsing techniques. It parses the syntax of the SQL query to gather the types of arguments. It detects malicious code when the user input does not match with the registered data type. This tool produced no FPs, no FNs, had low runtime overhead, and could be applied to web applications written in different languages. Its main disadvantage is the need to rewrite the SQL statements that were dynamically generated.

The approach proposed by Balzarotti et al. [83] first uses SA to identify built-in and custom protection routines. Then, to reduce the number of FPs, it uses DA to confirm the existence of potential vulnerabilities in these routines. They simulate the effect of the routine on the input by executing the code with different test inputs, containing various types of attack strings. This approach is limited to applications using such protection routines, which misses the vulnerabilities due to nonexistent data protection.

Lee et al. [139] combined both SA and DA for detecting SQLi attacks. They started by using SA to obtain the structure of the query, after removing the value of the attributes involved in SQL queries (e.g., values enclosed within quotes, or values followed by the equal “=” character). Afterwards, at runtime, the dynamically generated query is captured. Attacks are detected during runtime after comparing the syntactic structure of the queries with the predetermined one. The advantage of this approach is that the algorithm is capable of detecting the attacks at constant time. The results showed that the proposed method is very effective and simpler than other methods, such as AMNESIA and SQLCheck, which cannot detect stored procedure attacks.

Tripp et al. [144] uses SA and DA for detecting DOM-based XSS in the client side of web applications. They used DA (crawling) to collect DOM information. Instead of using an explicit modeling of DOM APIs, their SA used concrete DOM values that were obtained during DA. The evaluation results shown that the hybrid approach improves the SA precision over six times when compared to its purely SA counterpart. This hybrid approach was implemented as a JavaScript HA tool and was integrated into the IBM AppScan Standard Edition security web application scanner.

Alhuzali et al. [145] proposed an approach combining DA that is guided by SA techniques in order to automatically identify vulnerabilities and build working exploits taking into account the dynamic features and the navigational complexities (e.g., dynamically generated forms and links that may drive the navigation of the web application to vulnerable SSs) of modern web applications. The approach was implemented in a tool called NAVEX and has two major steps: *Vulnerable Sink Identification* and *Concrete Exploit Generation*. NAVEX first builds a graph model of each module’s code, then it discovers the paths that contain data flows between EPs and SSs. Finally, it uses symbolic execution to generate a model of the execution as a formula and constraint solving to determine which of those paths are potentially exploitable. According to the evaluation results, the first step reduced FPs by 87% on average. Generating the concrete exploits requires modeling the whole application. First, a dynamic execution step, using a web-crawler and a concolic executioner on the server-side, creates a navigation graph (i.e., sequence of HTTP requests) that captures the possible sequences in which application modules can be executed. Next, the navigation graph is used to discover the execution paths to only those modules that contain the vulnerable SSs. Finally, the exploit is generated and consists in sequences of HTTP requests with concrete parameters and respective values. The evaluation results show that the tool outperforms other approaches and scales to very large applications and to multiple classes of vulnerabilities. The evaluation results for SQLi shows that NAVEX was able to generate concrete exploits for 68% of the SQLi exploitable sinks. The dataset used to evaluate the tool was not characterized in terms of vulnerabilities (i.e., the number of vulnerabilities in the applications is unknown). Despite the ability of the tool for automatic vulnerability detection and exploit generation, its vulnerability detection coverage was not evaluated.

Balzarotti et al. [83] developed a tool called **Saner** that combines **SA** and **DA** techniques to identify faulty sanitization procedures (incorrect or incomplete) in PHP web applications that can be bypassed by an attacker. The component of **SA** is based on the open-source tool **Pixy** [81]. It provides information (**DFG**) about the existence of data flows between **EPs** and **SSs**. The **DA** phase for each **DFG**, identified as suspicious in the **SA**, examines all program paths from **EPs** to **SSs**. Then, it attempts to confirm the existence of potential security vulnerabilities simulating the effect of the program operations with generated inputs that can bypass the sanitization routines and reach the **SS**. The evaluation results showed that the proposed approach provided a method to reduce the **FPs** that are generated when a tool conservatively considers all custom sanitization routines to be ineffective, and **FNs** if the tool takes the opposite approach of considering all sanitization routines to be secure.

Yannis Smaragdakis [146], used **SA** to direct generation of test cases. Csallner et al. [147] proposed a series of tools to generate the test case with the assist of **SA**. Rao et al. [105] proposed an approach that combines tainted **SA** with penetration testing to detect input validation vulnerabilities (e.g., **XSS**, **SQLi**, etc.) in web application. The approach performs automatic penetration testing by leveraging the information obtained from **DA**. Monga et al. [148], proposed an approach and developed a tool (**Phan**) that works directly at the Zend bytecode level. They statically analyze PHP bytecode searching for dangerous code statements, and then only these statements are monitored during the **DA** phase to reduce the run-time overhead [149].

Xincheng et al. [150], proposed a hybrid analysis method consolidating **SA** and **DA** for detecting malicious **JavaScript** code that works by first conducting syntax analysis and dynamic instrumentation to extract internal features that are related to malicious code, and then performing classification based detection to distinguish attacks. In addition, based on code instrumentation, they proposed a new method which can deobfuscate part of obfuscated malicious **JavaScript** code accurately. The approach was implemented as browser plug-in called **MJDetector**. Evaluation results, based on 450 real web pages, showed that the method can detect malicious **JavaScript** code and de-obfuscate the obfuscation effectively and efficiently. In particular, **MJDetector** can distinguish **JavaScript** assaults in current website pages with a high precision of 94.76% and de-jumble muddle code of explicit sorts with 100% exactness though the gauge strategy can just identify with 81.16% exactness and has no limit of de-obscurety.

Park et al. [18] proposed an intelligent vulnerability analysis technique using risk evaluation. The method is based on existing **SA** and **DA** techniques and in an interaction analysis to increase detection accuracy of the static and the dynamic analysis. The authors developed a prototype analysis tool to test the vulnerability detection ability of the approach. The data set used for assessment includes 15,277 **Java** source file from the Juliet Test Suite ([114]). The evaluation was made up with both static and dynamic modules: two **SAST** tools (**PDM** and **FindBugs**) and seven Dynamic Application Security Testing (**DAST**) tools (**ZAP**, **Peach puzzer**, etc) were used. They found that the mix usage of **PDM** and **Peach puzzer** was the most superior over all other mix of techniques. The authors compared the proposed technique (**PMD** + **Peach puzzer**) with **PDM** and **Peach puzzer** for the Top 10 of **SANS** 25. The results showed that the proposed method is better in the number of vulnerabilities detected and detection accuracy, except for the class of vulnerability “*Use of hard-coded critical information*”, such as a password or cryptographic key.

2.6. Benchmarking

The most common method to assess and compare the performance of alternative tools is to run them with a set of representative test cases and compare the results. A standard process for doing this task is called a *benchmark* [151] [152]. **Benchmarks** are standard procedures that allow assessing and comparing different systems or components according to specific characteristics, such as performance, dependability, and security [151]. A **performance benchmark** is a test that measures the performance of a computer system or a component on a well-defined task or set of tasks. The task or set of tasks is normally defined by a *workload* and the measures are specific of each benchmark. A set of procedures and rules specifies the way the test must be conducted to reach valid benchmark results [153].

The key aspect that distinguishes benchmarking from other experimental evaluation techniques is that a benchmark is a standardized procedure that can be used to rigorously evaluate and compare the performance of different systems or components in a given domain, using well-characterized benchmark workloads, to determine the strengths of each system or component or to provide recommendations regarding suitable choices of systems and components for an analysis. However, benchmarking studies must be carefully designed and implemented to provide accurate, unbiased, and informative results [154].

A benchmark typically includes three main components [155]:

- 1) **Workload**: which is a set of representative test cases for the tools under benchmarking.
- 2) **Metrics**: to compare how the tools under benchmarking fit their purpose.
- 3) **Procedures and rules** for the benchmark execution.

To be accepted, any benchmark should fulfill a set of key properties: *representativeness*, *repeatability*, *nonintrusiveness*, *scalability*, *portability*, and *simplicity of use* [151], [156]. Lu et al. require a benchmark to be *representative*, *diverse*, *portable*, *accessible*, and fair for selecting suitable test cases for their workload (i.e., *customizability*). They also propose additional criteria to measure *usability* (reliance on manual effort and hardware, reporting and ease of investigating findings) and *overhead* (time for setting up and running the analysis, time and costs for training) [157].

The workload strongly determines the results, so it should be representative of all applications [155]. The perfect workload is a set of large production software, developed according to typical industry practices and whose vulnerabilities are all identified [158]. The high variety of applications constructed with heterogeneous components and the diversity of vulnerability classes make it unfeasible to define a benchmark for all tools in all situations. Therefore, the workload should be specifically built and configured for a particular application domain [159]. Users should be able to generate customized benchmarks that are tailored to their codebases and bug distribution expectations: one fixed benchmark does not suit all [160]. Unfortunately, the selection of a set of representative applications in a given domain is still a difficult task and creating them would consume immense resources.

Delaitre et al. [158] discuss three desirable characteristics for these workloads. First, **statistical**

significance: the test cases should be large enough (up to millions lines of code) to yield statistical significance. It is obtained through the size and diversity of the test cases (e.g., production software). Additionally, the tests need to have a sufficient number and diversity of vulnerabilities to achieve statistical significance. The results must demonstrate all the capabilities of the tools and in different instances (e.g., using test cases without vulnerabilities that a tool find very well). If some features remain unexposed, the generalization would be inaccurate. Second, **ground truth**¹²: we must know all the weakness **location**¹³ in the test cases. This enables faster tool warning evaluations and, more importantly, the identification of undetected vulnerabilities (FNs). The test case should reflect actual fixes made by real-world programmers to repair real vulnerabilities. Third, **relevance**: test cases should be close to those used in the industry. This means, software used in production environments and developed according to industry standards.

The selection of reference workloads is a critical design choice. Thus, the workload should ensure the following properties [155]:

- **Representativeness**. The workload should be typical of the domain in which the benchmark will be applied, because the benchmark results should provide relevant information to the users in the context of their planned use. This is influenced by the size and diversity of the test cases [158].
- **Comprehensiveness**: the workload should be able to exercise all the important features typically used in the target domain. Features should be balanced according to usage in real cases.
- **Focus**. The workload should be centered on characterizing the targets under benchmarking. Three criteria should be considered: *statistical significance*, *ground truth* and *relevance*.
- **Configurability**: users should be able to customize the workload considering their requirements, like security and budget.
- **Scalability**: the workload should increase or decrease in number and complexity of test cases, preserving the relation with the real application scenario.

There is no consensus on the metrics to use for evaluating the effectiveness of **SAST** tools (*Coverage*, *Precision*, *Recall*, *F-Measure*, *Discrimination*, etc.). As mentioned before, Delaitre et al. [158] identified three test case characteristics required to calculate such metrics: statistical significance, ground truth, and relevance. However, in practice, test cases respecting all these characteristics do not exist (or are not publicly available), and creating them is difficult due to the amount of effort that would be required. What we can find are test cases combining two of the characteristics: software with Common Vulnerability Enumeration (**CVE**) (relevance and ground truth), production software (statistical significance and relevance), and synthetic test cases (statistical significance and ground truth) [158].

Several studies survey the performance of **SAST** tools. For example, Kupsch and Miller [161] compared the results of two commercial **SAST** tools with an in-depth manual vulnerability assessment. The **SAST** tools just found a few of the several vulnerabilities discovered in the

¹²Ground truth - Knowledge of all vulnerabilities in a test case, including their location in code and vulnerability class.

¹³Location - A representation of a site, e.g., by file name and line number in source code.

manual assessment and missed many vulnerabilities requiring a deep understanding of the source code. Li and Cui [162] provided a technical description of seven open source **SAST** tools, describing their experience with three of them in terms of **FP** and **FN** rates on their own test code. They found that each tool has different advantages in finding different classes of vulnerabilities and all tools reported **FPs**. Emanuelsson and Nilsson [163] described three commercial tools, providing case studies on their evaluation at Ericsson. Such comparisons provide valuable insights for security researchers, but cannot be easily replicated.

Johns and Jodeit [164] introduced a common methodology for systematic evaluation of **SAST** tools using a benchmark composed of small programs that contain artificially injected vulnerabilities. The methodology relies on fine-grained, targeted tests. These test cases probes for an clearly defined, singled-out characteristic (e.g., comprehension of language features) of the evaluated tool. However, the overall complexity of the resulting test cases is lower compared to real-life applications. Therefore, artificial vulnerabilities may differ from the real-world ones, and therefore the evaluation results may differ.

Pashchenko et al. [165] proposed an approach (Delta-Bench) for the automatic construction of benchmarks for **SAST** tools based on the difference between vulnerable and fixed versions in free and open source repositories. The tool output, after analyzing a vulnerable version, would likely contain many alerts not related to the analyzed vulnerability. Such alerts should be also present in the tool output on a fixed version. Hence, the alert subtraction may significantly decrease the amount of irrelevant alerts. They considered such alerts as **FPs**, since the approach focused only on one vulnerability at a time. Unfortunately, the number of vulnerabilities in free and open source repositories is unknown. Therefore, the test cases are limited in terms of ground of truth and it is not possible to evaluate the overall capabilities (e.g., *Recall*) of the tools. In fact, the paper compares two tools to demonstrate the methodology without making claims about the overall performance of these tools.

Higuera et al. [166] proposed an approach to design a benchmarking for the evaluation of **SAST** tools capable of detecting software vulnerabilities, considering the **OWASP** Top Ten project vulnerability categories. They defined a test suite composed by 209 test cases selected from the **SAMATE** Juliet benchmark with different vulnerability types (e.g., **SQLi**) in each vulnerability **OWASP** Top Ten category (e.g., Injection). The authors do not mention the methodology used to select these test cases. Based on the vulnerable test cases, they generated 439 non-vulnerable variants of the testes cases, by fixing the vulnerability in the **EP** or/and in the **SS**. The approach was tested using seven **SAST** tools, five leader commercial and two open source, searching for vulnerabilities against the test cases. For evaluating the **SAST** tools, a panoply of metrics were calculated by vulnerability class and by **SAST** tools, providing precise information to practitioners selecting a tool. For ranking the **SAST**, they proposed a set of metrics to classify them according to three different degrees of web application criticality: *Recall* for business-critical, *F-Measure* for heightened-critical and *F_{0.5}-score* for non-critical. As alternative to *Recall*, they proposed to use *F_{1.5}-score* metric as it allows to reward the tools with better *Recall* than *Precision* metric. The results in terms of **TP** rate and **FP** rate varies widely both between **SAST** tools and class of vulnerability suggesting selecting a **SAST** tool by class of vulnerability. It is important to emphasize that one open source tool performed similar to the commercial tools and the other performed poorly.

Two public well known benchmarks for **SAST** tools are the **BSA** [167] from **OWASP** and the **SAMATE** project [117] from **NIST**. Through the development of the tool functional specifications [168], test suites and tool metrics, the **SAMATE** project establishes a methodology to understand the capability of **SAST** tools against a set of vulnerabilities. **SAMATE** promoted a recurring large-scale public event named Static Analysis Tool Exposition (**SATE**), designed to advance research in **SAST** tools that find security-relevant vulnerabilities in source code. The first event was conducted in 2008 and it is now in the sixth edition [169]. One of the main outcomes of this project is the creation of the **SARD**, which contains test suites for **SAST** tool comparison. **SARD** is a growing collection of almost two hundred thousand test programs with documented vulnerabilities. A variety of test cases are inspired on real applications, others on applications specifically developed for the benchmark and also on code written by students. The programs are in **C**, **C++**, **Java**, **PHP**, and **C#**, and cover over 150 classes of vulnerabilities. The metrics used to evaluate the tools are the False Positive Rate (**FPR**), *Precision* and *Recall*. The **BSA**, from **OWASP**, is a free and open test suite to evaluate the speed, coverage, and accuracy of automated **SAST** tools and services [167]. The workload contains about 23,8K **Java** test cases that are fully runnable and exploitable, including 11 classes of vulnerabilities. Each category comprises test cases with and without vulnerabilities. Instead of real applications, the test cases are small pieces of code with less than 100 lines, derived from coding patterns observed in real applications.

Evaluating the effectiveness of **SAST** tools using the **SAMATE** and the **BSA** benchmarks requires four steps: 1) running the **SAST** tools for detecting vulnerabilities in the synthetic workloads, 2) converting the results of the **SAST** tools to a common format, 3) comparing the results with the expected ones, and 4) computing the chosen evaluation metrics. The main limitation of both **SAMATE** and **BSA** is the synthetic workload, which is composed mainly by simple small test cases with few programming constructs, that may not be representative of production code, limiting the validity of the results in real conditions [158]. Also, the evaluation procedure does not consider the specific characteristics of the scenario where the tools are to be used. This contrasts with the reality, where applications are large and complex. Thus, with these test cases, it is very difficult to evaluate the real effectiveness of the **SAST** tools.

The metrics proposed by the **SAMATE** for evaluating the tools are *Precision*, *F-Score* (also called *F-Measure*), *Recall* and *Discrimination Rate (DR)*. The **DR** is applied to a pair of test cases: the bad and the good. The **bad test case** has a vulnerability and the **good test case** is essentially the bad test with the vulnerability fixed. While every **TP** counts when calculating recall, thus increasing the metric, for the **DR** a **TP** counts if the tool reports a vulnerability in the bad test case and does not report a **FP** in the good test case [113]. Since **DR** is applied to pairs of test cases, we would need a vulnerability free version of each test case (i.e., with all vulnerabilities fixed) to calculate it. However, for many test cases there is no fixed version available, so it may be difficult to compute the **DR** metric.

The Benchmark for Security Automation (**BSA**) established a scientific way to evaluate and compare tools. It defined a *single metric* called Benchmark Accuracy Score (**BAS**) which is equivalent to the *Informedness metric* normalized to the range $[-100, 100]$ [167]. It is based on the confusion matrix (**TP**, **FP**, **FN**, and True Negative (**TN**)) and is essentially a Youden's Index (J), which is a standard way of summarizing the accuracy of a set of tests with two classes (dichotomous) [167]. Youden's index evaluates the ability of the **SAST** tools to avoid failure (**FNs**

and FPs) and is defined using the Equation 2.1 as:

$$J = \textit{Sensitivity} + \textit{Specificity} - 1 = \textit{TPR} - \textit{FPR} \quad (2.1)$$

The *Sensitivity*, *Specificity*, **TPR** and **FPR** are defined as:

- **Sensitivity:** True Positive Rate (**TPR**) or *Recall* is the rate at which the tool correctly reports real vulnerabilities.

$$\textit{Sensitivity} = \textit{TPR} = \frac{TP}{P} = \frac{TP}{TP + FN} \quad (2.2)$$

- **Specificity:** or True Negative Rate (**TNR**) is the rate at which the tool correctly reports fake vulnerabilities as such.

$$\textit{Specificity} = \textit{TNR} = \frac{TN}{TN + FP} = 1 - \textit{FPR} \quad (2.3)$$

- **FPR:** the rate at which the tool incorrectly reports fake vulnerabilities as such.

$$\textit{FPR} = \frac{FP}{N} = \frac{FP}{FP + TN} \quad (2.4)$$

BAS is the normalized distance from the “guess line” (see Figure 2.14) and it is calculated as following:

$$\textit{BAS} = (\textit{TPR} - \textit{FPR}) \quad (2.5)$$

The **BSA** established chart plots (scorecard) for visualizing the performance of a tool or tools. Figure 2.14 illustrates an example for four tools. The chart include a slope and one diagonal random guess line. It means that a tool on that diagonal reported the same rate of **TPs** and **FPs**, and its score is zero. The left-up corner of the chart represents an ideal tool (no **FPs** and 100% of **TPs**) and the right-bottom corner the worst. This makes a lot of sense, since **FPs** cost a lot of time to validate and reduce the acceptance of a tool. Therefore, going up is good because the tool is reporting **TPs** and going to the right is bad because the tool is reporting **FPs**. On the chart, the **BAS** metric is the normalized distance from the point (**TPR**, **FPR**) down to the diagonal line. The chart can include the results of several tools by vulnerability class or the average of all vulnerability classes of the tools to provide an overall rank of the tools.

In addition to the workload issues of existing works, another limitation of these projects is the use of the same metrics independently of the environment where the vulnerability detection is going to be performed (projects have specific goals and constraints regarding criticality and budget). We aim to improve these aspects by using a representative set of real web applications with real vulnerabilities, and to use different evaluation metrics to rank the tools according to the scenario considered.

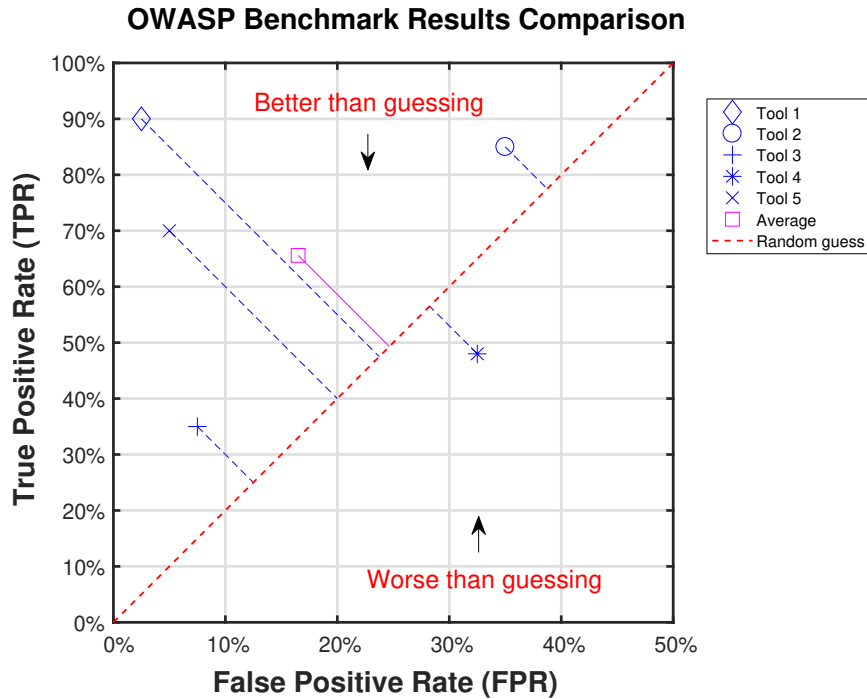


Figure 2.14.: OWASP Benchmark Results Comparison.

2.7. Conclusion

WordPress is the most widely used [CMS](#) adopted by businesses of all sizes and everyday website owners. WordPress and its plugins are developed in [PHP](#) which is, by far, the most popular server-side programming language used in web applications. WordPress applications have been under attack exploiting mostly plugin vulnerabilities, like [SQLi](#) and [XSS](#). Vulnerability detection in WordPress plugins is, therefore, an important matter that should be better studied.

Vulnerability detection approaches are commonly divided into three wide classes: [SA](#) or white-box analysis, such as source code review; [DA](#) or black-box analysis, such as penetration testing; and [HA](#) or gray-box analysis, which is a combination of white-box analysis and black-box analysis. Each technique has its strengths and weaknesses: [SA](#) is usually applied to the source code of the application early in the [SDLC](#), having a good coverage but many [FPs](#); while [DA](#) is applied later because requires the application running, having few [FPs](#) but a poor coverage. Existing [HA](#) joins [SA](#) with [DA](#) using different approaches as a way combine their strengths while minimising their weaknesses. Current automated security analysis and testing tools ([SAST](#)) have important limitations, such as failing to report vulnerabilities while reporting many [FPs](#), depending on the target project. We need means to know which [SAST](#) is better fitted to a specific project.

The most common method to assess and compare the performance of alternative tools is to run them with a set of representative test cases and compare the results. A standard process for doing this task is called a benchmark. Benchmarking can also have a beneficial effect on aspects needed to support continuous improvement, such as: raised awareness about performance and greater openness about relative strengths and weaknesses; and better understanding of the

“big picture” and gaining a broader perspective of the interplay between vulnerabilities and FP reported that facilitate the selection of good tools for a specific project. Currently available SAST tools benchmarks are very limited, being the most well-known efforts the SAMATE project from NIST and the OWASP Benchmark for Security Automation (BSA). Besides not producing true to life results, these benchmarks also lack the ability to be tailored to a specific context (e.g., critical or non-critical applications), which may affect the relevance of the results. The design of new benchmarks for SAST tools, like those we present in this work, helps to fill the gap.

A Security Analysis Tool for OOP Web Application Plugins

Source code review is a resource intensive task that is only feasible if supported by automated tools. There are several tools that can be used, but the vast majority of plugin developers cannot afford the expensive commercial source code analyzers. Although they can use free tools, like RIPS or Pixy a key limitation of these tools is the absence of capabilities to analyze OOP code, which is nowadays largely used for developing CMS applications [170]. Another drawback is the lack of knowledge about the CMS framework for which the plugin is being developed. For example, when analyzing the plugin, such tools are not aware of input and output vectors and of filtering functions included in the API of the CMS framework. These limitations lead to vulnerabilities being left undetected and, at the same time, to the generation of many false alarms.

Since version 5.0, PHP implements several OOP features like *classes*, *objects*, *properties*, *methods*, *inheritance* and *override* of methods. Coping with OOP is a very important matter, since plugins can access OOP code, even if they are developed using procedural programming. This happens because WordPress is developed using OOP, and plugins need to use the methods and attributes of existing WordPress objects. Moreover, some of these methods retrieve data from potentially untrustworthy sources. All OOP vulnerabilities we found are, indeed, related with WordPress objects and method calls.

This chapter proposes a methodology for detecting vulnerabilities in PHP source code and PHP plugins. The methodology is a follow-up of a project whose development was requested by Automattic, the developer of WordPress [171], with the goal of improving the security of a number of plugins. To demonstrate the feasibility of the proposed methodology we developed phpSAFE¹, a free SAST tool for PHP based plugins able to detect XSS and SQLi vulnerabilities.

To evaluate phpSAFE we compare its ability to detect plugin vulnerabilities with two well-known

¹<https://github.com/JoseCarlosFonseca/phpSAFE>

free tools, RIPS and Pixy. The three tools are used to detect vulnerabilities in 35 widely used WordPress plugins. We also study how the tools behave regarding the evolution of plugins over time by considering two versions of each plugin: one from 2012 and another from 2014. The data from the 2012 version was obtained from a previous work done by Fonseca et al. [170] that studied the effectiveness of the static analysis tools RIPS and Pixy in detecting WordPress plugin vulnerabilities.

The outline of this chapter is as follows. Section 3.1 presents the proposed detection approach and the phpSAFE tool. Section 3.2 details the approach followed to evaluate phpSAFE and Section 3.3 discusses the results. Finally, Section 3.4 concludes the chapter.

3.1. Detection Approach and the phpSAFE Tool

Figure 3.1 illustrates the methodology proposed for vulnerability detection in PHP code. It is based on tainted analysis and includes four stages: 1) configuration, 2) model construction, 3) analysis, and 4) results processing. Although the methodology can be seen as generic, to demonstrate feasibility we developed the SAST phpSAFE tool, focusing on SQLi and XSS, as these are two of the most important vulnerabilities in web applications nowadays. The tool was developed from the ground up with OOP and plugin security in mind, thus including OOP concepts like objects, properties and methods. It is also ready to detect vulnerabilities in plugins, including those developed using OOP. The stages of the proposed methodology and their implementation in phpSAFE are detailed in next sub-sections.

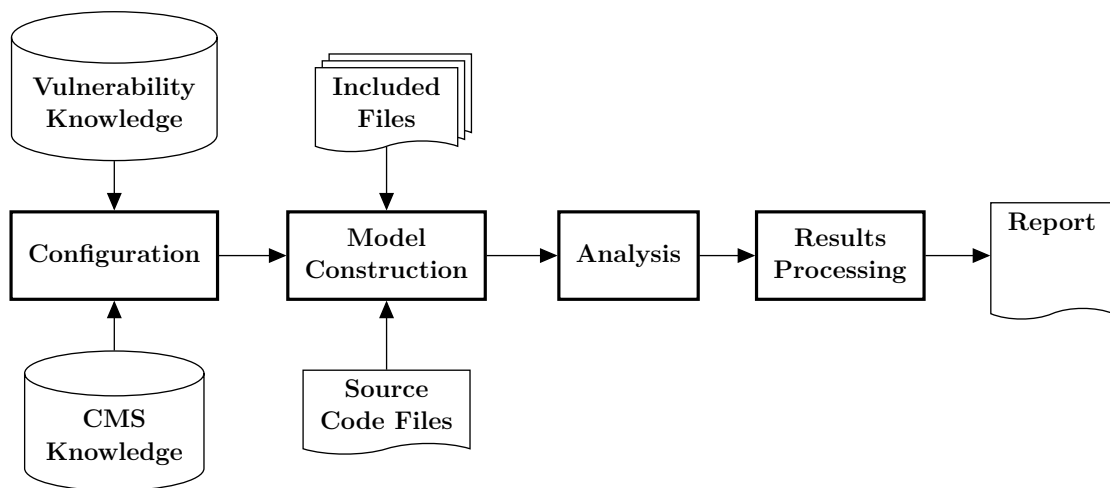


Figure 3.1.: phpSAFE architecture

The phpSAFE tool is itself a web application developed with PHP object oriented programming. Thus, the only requirement to run phpSAFE is a local web server with the PHP interpreter enabled and a web browser. The interface of phpSAFE allows the end-user to specify search and output options, and performs vulnerability scanning in PHP applications and plugins. The output of the analysis is presented in a web page that helps reviewing the results, including the vulnerable variables, the entry point of the vulnerability in the source code PHP file, the flow of the vulnerable

data from variable to variable, the visualization of the source code with the entry points, data flow of the vulnerable variables and **SSs** highlighted, etc.

As **phpSAFE** is developed in a **OOP**, its methods become accessible through the instantiation of a single PHP class **API** called **PHP_SAFE**, which receives as input the PHP file to be analyzed and delivers the results in the properties of the object instantiated from the **PHP_SAFE** class. In addition, the class provides a set of methods to produce specific results formatted in **HTML** stored in PHP strings or **HTML** files. Thus, it is prepared to be easily integrated with the software development process of other PHP projects. For example, the use of **phpSAFE API** can be part of the **SDLC** of a company, it can be used to automate the process of analyzing a large quantity of PHP scripts residing in different locations, it can be tuned to produce and store the results in other formats (e.g., Comma Separated Values (**CSV**) and **XML**) or distribute them over the network, etc. This integration ability is easily achieved by including the **phpSAFE API** in a PHP project.

3.1.1. Configuration Stage

The configuration stage consists of specifying information regarding vulnerabilities and the programming language environment. The configuration is made through parameters stored in external files and allows users to configure it according to the goals of the static analysis in different stages of the **SDLC** of each project. In this stage, the configuration data is loaded, containing the list of vulnerabilities (e.g., **SQLi** and **XSS**) correlated with the PHP language functions, and the target **CMS** framework specific functions that may have an effect in these vulnerabilities. In the configuration, these functions are organized in four main sections:

- 1) **Potentially malicious sources**: the Entry Points (**EPs**) of the attack.
- 2) **Sanitization and filtering**: the functions used by the target web application to prevent attacks.
- 3) **Revert functions**: functions that revert the actions of the sanitization and filtering functions, therefore allowing the attack.
- 4) **Sensitive output functions**: are the **SSs** where the attack manifests itself.

phpSAFE is deployed with a default configuration that is ready for detecting generic **XSS** and **SQLi** vulnerabilities, as well as for plugins of the WordPress framework. This solution, out-of-the-box, has the advantage of allowing the immediate use of the tool to analyze PHP code, either from applications or WordPress plugins without requiring further configuration. However, this ability can be easily extended to other **CMSs**, by adding their input, filtering and sink functions to the configuration files organized in categories and subcategories.

The generic **XSS** and **SQLi** functions in these configuration files are based on the default configurations of the **RIPS** tool [51]. Additionally, **phpSAFE** configuration files contain WordPress specific functions and class methods related with **XSS** and **SQLi**. It is here where data of other **CMSs** can be easily added to the configuration. In fact, this is what it takes for **phpSAFE** to be able to analyze plugins from other **CMSs**.

3.1.2. Model Construction Stage

In this stage, a lexical and semantic analysis is performed based on the Abstract Syntax Tree ([AST](#)) of the PHP source code. The [AST](#) is obtained using the PHP function `token_get_all` that splits the PHP code into tokens. Each token can be an array with three items or a string:

- The array has the token identifier, the value of the token and the line number of the PHP script (e.g., `[310, $_POST, 11]`);
- The string represents code semantics (e.g., `“;”`).

An [AST](#) is built for each PHP file being analyzed. This [AST](#) is cleaned by removing comments and extra white-spaces. As the source PHP file may include other PHP files recursively, all of them must be analyzed in order to obtain the complete [AST](#). To speed up the analysis and the ability to cope with plugin code, information is collected from the [AST](#) about all user-defined functions and their parameters, all the called functions, among other relevant data. This allows, for example, obtaining the list of plugin functions that are not invoked from within its code. However, these functions should be parsed anyway, as they may be directly called from the main web application. This ability to analyze all the functions, even those not directly called from within the plugin, is a very important feature of security tools targeting plugin code.

Parsing of each token requires distinguishing between variables and properties, functions and methods, and act accordingly. For properties and methods the tool obtains the full name by adding the name of the object to which they belong through a backward search in the [AST](#) (by following the `T_OBJECT_OPERATOR` and `T_DOUBLE_COLON` tokens). Each property may (and will) then be parsed as a variable. The call to a method, including object creation with the PHP `new` construct, is parsed as a function by locating the source code of the called method inside the class.

3.1.3. Analysis Stage

The objective of this stage is to follow the flow of the tainted variables from the moment they enter the application/plugin (i.e., from [EPs](#)) until they reach the output (i.e., [SSs](#)). While the input is any `GET`, `POST`, `COOKIE`, database values, files, etc., the output may be the display of the variable in a web page, the storage of the variable in an Operating System ([OS](#)) file or the database, etc. During this process, the tainted variable may contaminate recursively other variables that should also be followed until they are finally outputted. On the other side, the malicious content of the variable may also be removed or neutralized using sanitization and filtering functions, preventing its exploitation.

The data flow history of each variable is stored in a multi-dimensional associative array `parser_variables`. This array contains everything needed to allow performing the taint analysis, like the variable name, source file name and line number, the dependencies from other variables, if it is an input or output variable, the filter functions applied, etc.

To gather the data needed to fill the `parser_variables` array, the tool follows the flow of the tainted variables. Next, it parses all the [AST](#) files previously created and makes decisions

based on code constructs like conditionals, loops, assignments, expressions, function or method calls, function or method returns, etc., which is done by following the path of the code, usually starting from the “main function”. Furthermore, it is able to parse plugins that do not have a “main function” or include functions that are never called directly from the plugin code. To reach 100% code coverage, all the functions should be analyzed, even those that are never called. To address this, it starts by executing an inter-procedural parsing of the functions that are not called from the source code of the plugin. Then it performs the inter-procedural analysis starting from the “main function” and follow the program flow from there. This way, every piece of code of the plugin is analyzed. The intra-procedural parsing goes through every token of the [AST](#) and parses it according to its nature.

3.1.4. Results Processing Stage

One of the objectives of source code security analysis is the identification of vulnerabilities so they can be fixed. `phpSAFE` provides several valuable resources to help developers in this task. Some of these resources are related to the variables (vulnerable variables, output variables and all the other variables), functions, PHP files included, tokens (the complete [AST](#)) and debug information. This data can be very useful in helping security practitioners to trace back the path of the tainted variables to the point they entered the system and locate the best place to fix the vulnerabilities found.

As an example of the tool interface, [Figure 3.2](#) shows a screen capture of the results of `phpSAFE` after analyzing the source code of the file `testSQLi.php` in [Listing 3.3](#). The results are organized in several items and showed in tables. The table *Vulnerable Variables* shows the vulnerable variables and includes several attributes about the variables such variable name, line in code and class of vulnerability. The *start_index* and *end_index* attributes indicates the position of the valuables in the [AST](#) and is under the item *Hide/Show File Tokens*. The *dependencies_index* allows to get data flows of the variable from the [SS](#) (`mysqli_query`) to the [EP](#) (`$_POST`). For example, in the table of the *Parser Variables* the variable `$sql` in the last line (6) has the following data flow: `$sql (6) → $sql(2) → $user(3) → $user(0) → $_POST['username'](1)`.

phpSAFE - PHP Security Analysis FOR EVERYONE

Home

Security analysis of the file

[C:\xampp55\htdocs\phpSAFE-master\test\testSQLi.php](#)

1 vulnerabilities found in 0.00 seconds!

Choose another file

Vulnerable Variables

Show Count: 1

#	index	name	object	scope	variable_function	exist_destroyed	code_type	input	output	function	file	line	tainted	vulnerability	start_index	end_index	dependencies_index	variable_filter	variable_revert_filter
0	6	\$sql		local	variable	exist	php code	regular	output	function	0	4	tainted	SQL Injection	22	22	2		

[Vulnerable Variables](#)

Output Variables

Show Count: 2

Parser Variables

Hide Count: 7

#	name	object	scope	variable_function	exist_destroyed	code_type	input	output	function	file	line	tainted	vulnerability	start_index	end_index	dependencies_index	variable_filter	variable_revert_filter
0	\$user		local	variable	exist	php code	regular	regular	function	0	2	tainted	unknown	1	1	1		
1	\$_POST['username']		local	variable	exist	php code	input	regular	function	0	2	tainted	unknown	3	6			
2	\$sql		local	variable	exist	php code	regular	regular	function	0	3	tainted	unknown	8	8	3		
3	\$user		local	variable	exist	php code	regular	regular	function	0	3	tainted	unknown	12	12	0		
4	\$result		local	variable	exist	php code	regular	regular	function	0	4	untainted	unknown	16	16			
5	\$connection		local	variable	exist	php code	regular	output	function	0	4	untainted	unknown	20	20			
6	\$sql		local	variable	exist	php code	regular	output	function	0	4	tainted	SQL Injection	22	22	2		

File Functions

Show Count: 1

Used Functions

Show Count: 1

Show All | Hide All | Default All

Show/Hide Parser Debug (0)

Show/Hide Vulnerable Variables (1)

Show/Hide Output Variables (2)

Show/Hide Parser Variables (7)

Show/Hide Files Functions (1)

Show/Hide Used Functions (1)

Show/Hide Files Include Require (0)

Show/Hide Files Tokens (1)

Show All | Hide All | Default All

Figure 3.2.: phpSAFE results for the code in Listing 3.3.

```

1 <?php
2 $user = $_POST['username'];
3 $sql="SELECT * FROM UserAccounts WHERE username='$user'";
4 $result = mysqli_query($connection, $sql);
5 ?>

```

Figure 3.3.: Source code of file testSQLi.php of project test.

3.2. Evaluation of phpSAFE

This section presents the experiments conducted to evaluate the performance of the phpSAFE tool and thus understand its strengths and weaknesses. Given the current web scenario with many plugin based web applications developed with PHP and OOP and the existence of other static code analyzer tools, there are three main questions that we are addressing in this evaluation:

- 1) How does phpSAFE performance compares with other free static analysis tools when analyzing open source plugins for an OOP developed web application, considering the most common and widely exploited vulnerabilities?
- 2) How does phpSAFE cope with the evolution of plugin code and vulnerabilities over a two-year-period of time, by looking at different versions of the same plugin?
- 3) Are plugin developers taking into consideration the disclosed vulnerabilities in subsequent versions of the plugins, even for vulnerabilities easy to spot and exploit?

To address the research questions we defined an experimental procedure based on five steps:

- 1) Selection of a widely deployed OOP web application with many open source plugins available. We selected WordPress, because it is developed in PHP and is the most widely used CMS [172], supporting the creation of web sites like TED, NBC, CNN, The New York Times, Forbes, eBay, Best Buy, Sony, TechCrunch, UPS, CBS Radio, etc. In 2014 there were millions of WordPress sites, and they account for 23% of the web [171]. Actually, in 2021, 42% of the web is built on WordPress².
- 2) Gathering web application plugins and two-years-old versions of those same plugins (from 2012 and 2014). In a previous work, Fonseca et al. [170] selected a set of 35 WordPress plugins, which is a reasonable number that allows both the execution of the experiments (including a manual verification of all the vulnerabilities reported by the static analysis tools), and that could be representative enough to obtain meaningful results. The 2012 versions of the plugins were analyzed by Fonseca et al. [170] in 2013 and the vulnerabilities found were then communicated to the developers [170]. To understand what actions were taken to mitigate those vulnerabilities, we analyzed the 2014 versions and compared the results.
- 3) Selection of well-known free security static analysis tools for web applications for comparison purposes. RIPS and Pixy are two of the most referenced PHP static analysis tools, although they are not ready for OOP analysis. They also have been subject of several scientific publications [93], [50], [170]. RIPS has only been developed as open source until 2014, and in 2016 released a commercial version able to fully analyze OOP code, but Pixy has not been updated since 2007. We focus on free SAST as both occasional developers and professional software houses wanting to speed up the development process and reduce cost tend to use free tools as much as possible.
- 4) Execution of phpSAFE and the other tools to search for vulnerabilities in the collection of plugins selected.
- 5) Analysis of the results. As each tool delivers the results in a specific format, we normalized

²<https://wordpress.com>

and merged all of them into a single repository. All the vulnerabilities reported by the tools were manually verified by a security expert looking for misclassification issues, although it was a labor intensive and time consuming task. This manual process was a very important step not only to guarantee the correct labeling of all the outputs of the tools, but also to obtain both an annotated collection of vulnerabilities of the plugins selected, which could be helpful in future comparisons of security tools.

Table 3.1 list the WordPress plugins used in the study. The plugins vary widely both in terms of number of Line of Code (LOC) and number of files. Overall the plugins contain 162,290 LOC and 789 source code files. To have an idea of their relevance, they are used in business, e-commerce, monetization, social networking, photo and video gallery, registration, admin, advertising, email, bookings, events management, newsletter, and document manager.

Table 3.1.: List of WordPress plugins.

#	Plugin	LOC	Files
1	all-in-one-webmaster 9.5	876	13
2	calendar 1.3.3	2693	1
3	content-slide 1.4.2	440	3
4	contextual-related-posts 2.0.1	1711	18
5	digg-digg 5.3.6	3850	13
6	easy-adsense-lite 7.43	3019	9
7	events-manager 5.5.3.1	25230	182
8	external-video-for-everybody 2.1.1	252	1
9	feedweb 3.0.6	4037	13
10	foursquare-checkins 1.6	362	10
11	funcaptcha 1.2.1	1608	8
12	ga-universal 1.0.1	248	3
13	jaspreetchahals-coupons-lite 2.8	1054	3
14	login-with-ajax 3.1.4	1116	9
15	mail-subscribe-list 2.1.1	354	3
16	mathjax-latex 1.3.3	339	2
17	montezuma 1.1.7	4862	55
18	newsletter 3.6.4	9103	102
19	occasions 1.0.4	525	3
20	paypal-digital-goods-monetization-powered-by-cleeng 2.2.16	2356	18
21	qtranslate 2.5.39	3489	9
22	securimage-wp-fixed 3.5.3	2999	6
23	simply-poll 1.4.1	808	15
24	social-media-widget 4.0.2	765	1
25	syntaxhighlighter 3.1.10	827	1
26	top-10 1.9.10.1	2152	12
27	trafficanalyzer 3.4.2	4051	34
28	underconstruction 1.12	573	3
29	user-role-editor 4.17.2	3097	19
30	videojs-html5-video-player-for-wordpress 4.5.0	372	3
31	wordpress-simple-paypal-shopping-cart 4.0.4	2216	10
32	wp125 1.5.3	688	6
33	wp-photo-album-plus 5.4.18	30295	75
34	wp-symposium 14.10	34847	117
35	xili-language 2.15.2	11076	9
Total		162,290	789

3.3. Results and Discussion

This section presents the results of the experiments, starting with an overall analysis and then going into the details. Results show that `phpSAFE` clearly outperforms other tools, and that plugins are being shipped with a considerable number of vulnerabilities, which tends to increase over time.

3.3.1. Overall Analysis

Table 3.2 depicts the global results obtained by executing `phpSAFE`, `RIPS` and `Pixy` with all plugins. For each tool there are two columns: one for the most recent version of the plugins (V.2014) and another for the older version (V.2012). The table presents the values of the chosen metrics: `TP`, `FN`, *Precision*, *Recall* and *F-Score*. The `FN` is needed to calculate the *Recall* and this implies knowing all the vulnerabilities present in the plugins. As mentioned before, due to the unfeasible efforts needed, we did not search the plugins for all the possible vulnerabilities using all possible means, like a thorough manual code review and using commercial tools. Therefore we considered as `FN` the vulnerabilities not detected by a tool, but were detected by any of the other tools. So, the value of the *Recall* metric is optimistic, although we expect it to be close to the actual cases.

Of the 35 plugins analyzed, 19 are developed in `OOP`. `phpSAFE` found 151 vulnerabilities related to the use of WordPress objects in 10 plugins of the 2012 version, and 179 vulnerabilities in 7 plugins of the 2014 version. `RIPS` and `Pixy` were not able to detect any vulnerability of this kind. In fact, results show that, `phpSAFE` is the tool that detects more vulnerabilities (`TPs`). It is followed by `RIPS` and `Pixy` which has the lowest detection value. The detection rate trend ranking is also followed by the other metrics (*Precision*, *Recall* and *F-Score*), for which `phpSAFE` also has the highest values followed by `RIPS` and `Pixy`. This means that `phpSAFE` is able to detect vulnerabilities more exactly (best *Precision*) and that it leaves less vulnerabilities undetected (best *Recall*) than the other tools. `phpSAFE` also has the best *F-Score* (or *F-Measure*, which represents the harmonic mean of precision and recall) among the tools. These results show that `phpSAFE` should be the tool chosen for all situations, from business-critical applications to the less critical scenarios. However, it still leaves vulnerabilities undetected, so other assurance activities should also be used.

This ranking of `phpSAFE`, `RIPS` and `Pixy` holds also true when considering 2012 and 2014 versions of the plugins. One of the reasons for the detection performance of `phpSAFE` is its ability to cope with `OOP` and its out-of-the-box configuration for WordPress plugins. Comparing the two versions of the plugins, it seems that `phpSAFE` and `RIPS` are up-to-date with current programming practices, as they detected more vulnerabilities in recent plugins. Conversely, `Pixy` detected less, maybe due to the lack of upgrades since 2007.

`phpSAFE` was the only tool able to correctly detect `SQLi` vulnerabilities, which is an odd result, since both `RIPS` and `Pixy` are also capable of detecting `SQLi`. This may be due to the small number of `SQLi` in the plugins analyzed, which may not have triggered the detection mechanism of these tools. An interesting result is the 115% increase in `XSS` detection by `RIPS` from the 2012

Table 3.2.: Vulnerabilities of 2012 and 2014 plugin versions.

		phpSAFE		RIPS		Pixy	
		V.2012	V.2014	V.2012	V.2014	V.2012	V.2014
XSS	<i>TP</i>	307	374	134	288	50	20
	<i>FP</i>	63	57	79	47	185	197
	<i>Precision</i>	83%	87%	63%	86%	21%	9%
	<i>Recall</i>	85%	68%	37%	53%	13%	4%
	<i>F-Score</i>	84%	76%	47%	65%	16%	5%
SQLi	<i>TP</i>	8	9	0	0	0	0
	<i>FP</i>	2	5	0	1	0	0
	<i>Precision</i>	80%	64%	-	0%	-	-
	<i>Recall</i>	100%	100%	0%	0%	0%	0%
	<i>F-score</i>	89%	78%	-	-	-	-
Global	<i>TP</i>	315	383	134	288	50	20
	<i>FP</i>	65	62	79	79	187	208
	<i>Precision</i>	83%	86%	63%	79%	21%	9%
	<i>Recall</i>	80%	66%	34%	52%	13%	3%
	<i>F-score</i>	81%	75%	44%	63%	16%	5%

to the 2014 version. This occurred because RIPS was able to detect vulnerabilities in some files of the 2014 versions that phpSAFE was unable to parse because these files had many includes and required a lot of memory. In fact, phpSAFE during the analysis stores in memory lots of data related with the source code and the vulnerabilities found. For example, to create the data flows of all variables are used arrays that stores the name of the variables for each link in the data flow consuming lots of memory. Replacing these arrays by trees can drastically reduces the memory and certainty that the tools performs better.

During the analysis process, we also observed that although phpSAFE and RIPS are able to detect vulnerabilities in functions that are not called from the plugin code, Pixy is unable to do so. Since this aspect may be common in plugins, as some functions are to be called from the main web application, this capability is a feature that all tools prepared for analyzing plugins should have.

3.3.2. Vulnerability Detection Overlap

Since no tool presents a 100% Recall, using more tools should allow detecting more vulnerabilities. Conversely, it is also likely that different tools may report some common vulnerabilities. In fact, we found such situations during the manual verification of the vulnerabilities. They are depicted in Figure 3.4 that shows a Venn diagram having the radius of the circles proportional to the number of vulnerabilities, providing a comparative visual image of the coverage of each tool.

Combining the results of all tools, we detected 394 distinct vulnerabilities in 2012 versions and 586 in 2014 versions. This is an increase of 51% in just two years, even though developers were informed about the vulnerabilities detected in 2012. In the diagram we can see that, although some vulnerabilities were reported by several tools (represented by the intersection of the circles), different tools also detected many different vulnerabilities. This confirms the well-known idea that there is no silver bullet to solve all security problems [173]. Furthermore, during the manual

verification, additional vulnerabilities were found (represented by an empty circle in the figure). As this was not done systematically we do not have this data accurately defined. However, the fact that there are vulnerabilities that were not detected by either one of the tools reinforces the need for performing other types of vulnerability detection analysis, besides using automated tools. Many researchers and practitioners also advise the use of other security practices like security training, manual code reviewing, black-box testing, etc. [174].

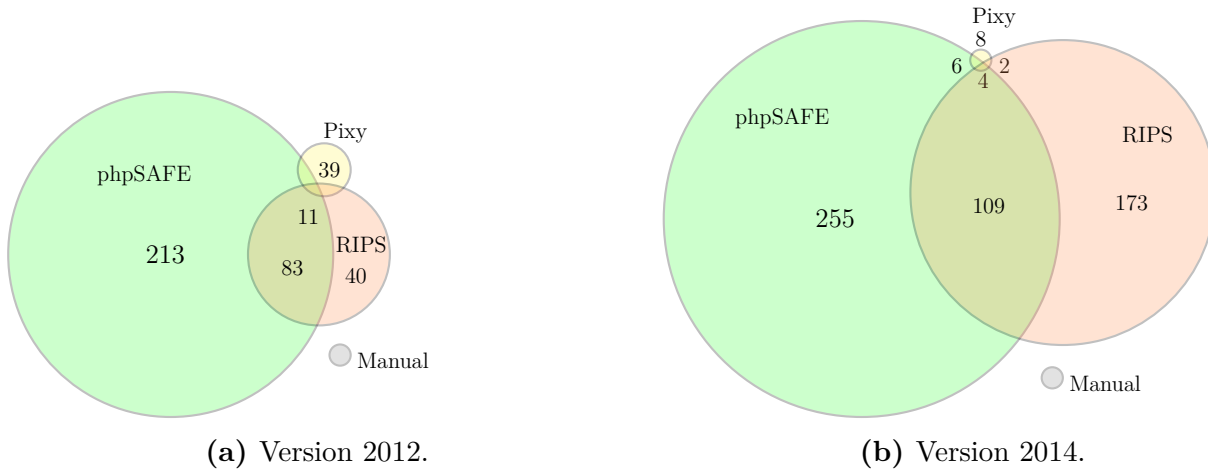


Figure 3.4.: Tools Vulnerability Detection Overlap.

Listing 3.1 shows a slice of code with a trivial XSS vulnerability in function `__wps__plugin_menu` at line 38. We removed blank lines, lines from 17 to 34 and lines from 40 to 7894 for simplicity. The vulnerability was reported by `phpSAFE` and `RIPS`. `Pixy` does not report a vulnerability because the tool does not analyzes code that is not called in the target source code. In fact, the function is called indirectly in line 7896 through the function `add_action`. It is important to emphasize that the developer used security protection (prepared statement with type cast to int for the parameter `$_GET['tid']`) for the `SS` in line 35 to prevent `SQLi`. However, the developer does not protect the code against `XSS` in line 38.

```

6 function __wps__plugin_menu() {
...
11 if (isset($_GET['action'])) {
13     switch($_GET['action']) {
15         case "post_del":
16             if (isset($_GET['tid'])) {
18                 if (__wps__safe_param($_GET['tid'])) {
...
35                 $wpdb->query( $wpdb->prepare( "DELETE FROM ".$wpdb->prefix."symposium_topics WHERE tid = %d", $_GET['
tid'] ) );
37             } else {
38                 echo "BAD PARAMETER PASSED: ".$_GET['tid'];
39             }
...
7895 if (is_admin()) {
7896     add_action('admin_menu', '__wps__plugin_menu');
7897 }

```

Listing 3.1: Slice of code from the file `menu.php` of the `wp-symposium.14.10`.

3.3.3. Inertia in Fixing Vulnerabilities

One of the quality assurance activities that should be done while maintaining software during its lifecycle is fixing bugs, giving priority to those that are more critical, like security issues. The vulnerabilities found in the 2012 version of the plugins were initially disclosed to the developers in November 2013 [170]. In the present study, we analyzed which of the vulnerabilities found in the 2014 version were among the ones previously disclosed in the 2012 version. Figure 3.5 plots a bar chart with the total number of vulnerabilities found in the 2012 and 2014 versions of the plugins. In the 2014 versions there are 337 new vulnerabilities and 147 vulnerabilities in the 2012 versions are not present in the 2014 version. We found that 249 (42%) of the vulnerabilities discovered in the 2014 version are among the ones discovered and disclosed to the developers more that one year ago (see Figure 3.5).

Figure 3.6 shows a pie chart for the classes of EPs of the vulnerability prevalence in the 2012 and 2014 versions of the plugins. From those, 190 (76%) are very easy to exploit, through simple GET, POST or COOKIE manipulation. This is a disturbing result that should raise the awareness of plugin developers, software maintainers of the CMS frameworks, of the site administrators, and of the end users.

Our results are in accordance with several security statistics in research studies and has not changed over several years. For example, software security adviser WhiteHat Security has estimated that the average time to correct critical cybersecurity vulnerabilities increased from 197 days to 205 days between April and May 2021 [175]. However, the data we used is not enough to perform a more in-depth analysis, which would require data across several years and a more representative set of applications.

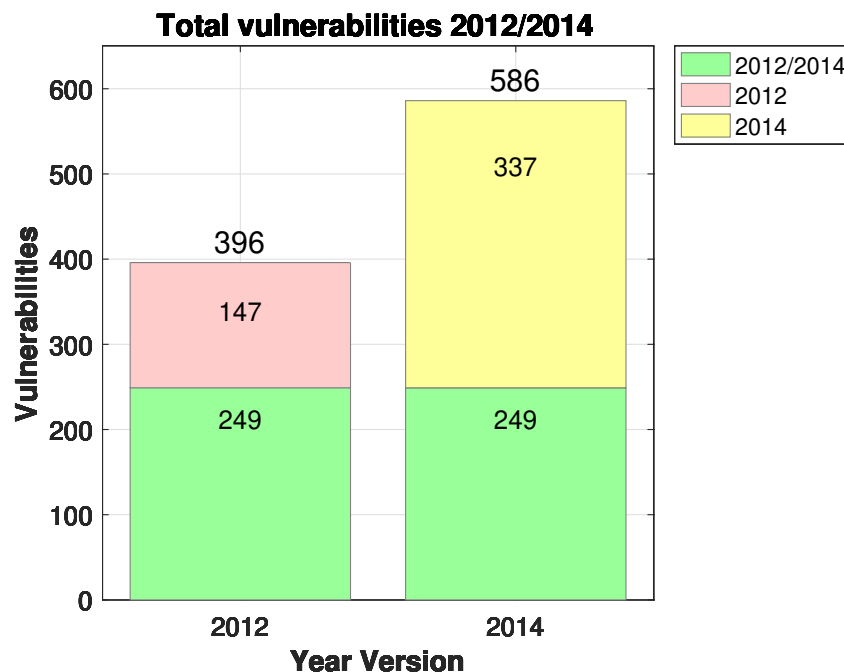


Figure 3.5.: Vulnerabilities of version 2012 and version 2014.

Vulnerability Prevalance 2012/2014

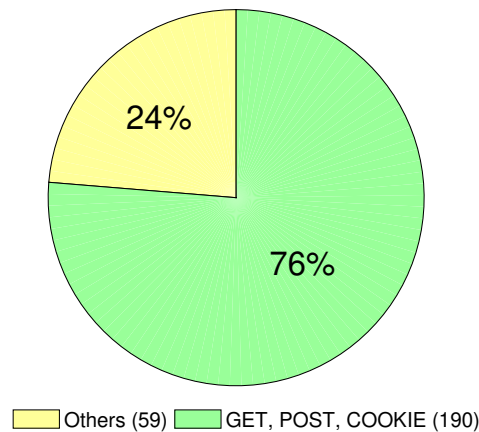


Figure 3.6.: Vulnerability prevalence between 2012 and 2014 by [EP](#).

3.4. Conclusion

In this chapter we presented `phpSAFE`, a source code vulnerability analyzer that is able to detect both [XSS](#) and [SQLi](#) vulnerabilities in plugins of applications developed in PHP with [OOP](#).

There are other free tools to search for vulnerabilities in PHP code, like `RIPS` and `Pixy`, but they are neither ready for [OOP](#) nor for analyzing plugins. As WordPress applications are so common, we evaluated `phpSAFE`, `RIPS` and `Pixy` with a set of 35 WordPress plugins, according to *Precision*, *Recall* and *F-Score* metrics. Due to its novel features, `phpSAFE` outperformed the other tools.

Results show that `phpSAFE` is able to detect more vulnerabilities than the other tools, with fewer false alarms. We also observed that both `phpSAFE` and `RIPS` deal well with the evolution of plugin code. A key observation is that plugins that are currently being used in thousands of WordPress installations have dangerous [XSS](#) and [SQLi](#) vulnerabilities and this number is increasing over the years. In fact, we discovered more than 580 vulnerabilities in the plugins analyzed, many of them very easy to detect and exploit. We also verified that developers did not fix many vulnerabilities even after knowing them for more than one year.

We used two versions of the 35 plugins to analyze how the tools cope with the evolution of the code. `phpSAFE` and `RIPS` did not show a relevant change in their detection performance, but `Pixy` had a significant decrease, possibly due to its lack of updates since 2007. Also, we were able to notice a 50% increase in the number of vulnerabilities in just two years. A more critical observation is that 40% of all the vulnerabilities found in the updated plugins were already present in the older version, even for those vulnerabilities that were disclosed to the developers more than one year ago.

The experiments also showed that using several tools allows increasing the number of different vulnerabilities detected, showing that this is a good direction to be researched. In [Chapter 5](#) we present several studies combining the results of diverse [SAST](#) tools as a way to increase the

overall number of vulnerabilities reported by these tools.

Future work includes the improvement of `phpSAFE`, mainly regarding performance, memory consumption and vulnerability coverage, along with the analysis of other [CMS](#) applications like Drupal or Joomla. Other enhancement is to extend the tool for detecting other classes of vulnerabilities.

Benchmarking Static Analysis Tools for Web Security

The selection of an appropriate [SAST](#) tool for a specific project is a difficult task as there are many [SAST](#) tools available, and each one has its own strengths and weaknesses. In fact, different [SAST](#) tools analyzing the same code report different results. The most common method to assess and compare the performance of alternative tools is to run them with a set of representative test cases and compare the results. A standard process for doing this task is called a *benchmark* [151] [152]. The key aspect that distinguishes benchmarking from other experimental evaluation techniques is that a benchmark is a standardized procedure that can be used to rigorously evaluate and compare the performance of different tools in a given domain, using well characterized benchmark workloads, to determine the strengths of each tool or to provide recommendations regarding suitable choices of systems and components for an analysis. Currently available [SAST](#) tools benchmarks are very limited, being the most well-known efforts the [SAMATE](#) project from [NIST](#) [117] and the [OWASP](#) Benchmark for Security Automation ([BSA](#)) [167]. Besides not producing true to life results, these benchmarks also lack the ability to be tailored to a specific context (e.g., critical or non-critical applications), which may affect the relevance of the results.

This chapter proposes an approach to design benchmarks for the evaluation of [SAST](#) tools that detect vulnerabilities in web applications considering different levels of criticality. Contrasting with [SAMATE](#) and [BSA](#), we propose the use of *workloads* composed by real applications that have known vulnerabilities (used to exercise the [SAST](#) tools, thus supporting their evaluation). This assures that [SAST](#) tools are tested considering the need to address both the complexity and the way real code is built, instead of processing much simpler synthetic code samples or test cases (as done by [SAMATE](#) and [BSA](#)). In fact, research shows that [SAST](#) tools perform better with synthetic test cases than with real software [158]. Additionally, by exploring the notion of *application scenarios*, our approach allows a better match of its outcomes with the requirements for the [SAST](#) tool operation. In particular, we consider four representative real-world usage scenarios, ranging from the development of business-critical to lower-quality web applications.

The use of application scenarios in the benchmark raises two fundamental challenges: *how should the SAST tools be ranked?* and, *how should the workload be created?* To rank the SAST tools we need several metrics, because no single metric is suitable to quantify all aspects of the performance of SAST tools in distinct scenarios [159]. Our approach relies on one main metric and a tiebreaker metric for each scenario, where the first is used to rank the tools and the second to decide eventual ties between two or more tools. To compose the workload, we consider a representative group of vulnerable applications for each scenario. Since this is very hard to attain (e.g., business-critical software is often kept secret) and has an associated level of subjectivity (e.g., there are different interpretations of what constitutes critical software), we propose a standard procedure to assign applications to scenarios based on their *code quality*. Generically, the assumption is that, *scenarios that are more stringent normally run software with better quality*. Therefore, we should assign applications with better quality to scenarios with higher criticality. The quality of the applications is measured using a quality model based on the ISO/IEC 9126 standard, relying on a set of source code metrics (e.g. the Cyclomatic Complexity Number (CCN)), which are related to non-functional requirements and can be obtained without running the applications.

The structure of this chapter is the following: Section 4.1 proposes a new approach for the definition of benchmarks for SAST tools and discusses the main components of such benchmarks. Section 4.2 presents an instantiation of the proposed approach. The goal of the benchmarking campaign presented in Section 4.3 is to compare and rank several SAST tools that search for SQLi and XSS vulnerabilities in WordPress plugins. Section 4.4 concludes the chapter.

4.1. Benchmarking Approach

Our benchmarking approach follows a specification-based style, where the specification defines the functions that must be achieved by the target SAST tools, the required inputs (workload consisting of representative vulnerable software) and the outcomes (vulnerability detection and metrics) [176]. Essentially, the idea is to run the target SAST tools using as input a set of real-world vulnerable software and, after gathering the vulnerabilities identified by the SAST tools and verifying their correctness, use a small set of metrics that summarize the detection capabilities of the tools to obtain a ranking, for each target scenario.

The high variety of applications constructed with heterogeneous components and the diversity of vulnerability classes make it unfeasible to define a benchmark for all SAST tools in all situations. Therefore, a benchmark should be specifically built or configured for a particular domain to allow making educated choices during the definition of the components [159]. In this work, defining the **benchmark domain** directly affects the workload and includes selecting the class of web applications (banking, social networking, etc.) and the classes of vulnerabilities (SQLi, XSS, etc.) to be detected by the target SAST tools. Also, the strengths and weaknesses of the workload depend on the balance of several criteria, often conflicting. Since no single workload can be strong in all criteria, there will always be a need for considering multiple workloads [176]. Therefore, our proposal is to define a set of workloads according to specific scenarios. Moreover, the workload should be built using a representative set of real software code with vulnerabilities.

The overall **SAST** tools benchmark architecture is illustrated in Figure 4.1. The approach is composed of four components:

- 1) **Scenarios**: requirements representing real contexts, with constraints according to the criticality level, where **SAST** tools will be used.
- 2) **Metrics**: used to characterize and compare the effectiveness of the **SAST** tools under benchmarking, in each specific scenario.
- 3) **Workload**: representative applications, with a set of vulnerabilities, to be used in each scenario. The classes of vulnerabilities (e.g., **SQLi**, **XSS**, etc.) should be representative of the target application domain.
- 4) **Procedures and rules**: description of the procedures and rules that must be followed during the benchmark execution using the workload. For each scenario, the benchmark produces a report with the ranking of the **SAST** tools under benchmarking, ordered by the relevant metrics.

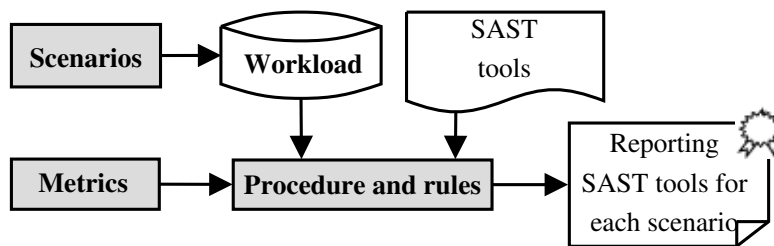


Figure 4.1.: General architecture of the benchmark.

The following sections discuss how to define each component.

4.1.1. Application Scenarios

An **application scenario** is a realistic situation of vulnerability detection that depends on the criticality of the application being tested and on the security budget available. Existing benchmarks have strong representativeness limitations, disregarding the specificities of the environment [164][117][167][165][166], where the **SAST** tools under benchmarking will be used, which may vary both in terms of development time and resources. Therefore, the test cases in the workload should reflect typical usage scenarios [177]. For example, users of benchmarks should be able to customize their evaluation by disregarding certain types of vulnerabilities [178]. The use of scenarios is challenging, in what concerns the benchmark metrics. In fact, to rank the **SAST** tools we need several metrics, that can be specific for each scenario, because no single metric is suitable to quantify all aspects of the performance of **SAST** tools in distinct application scenarios [159].

A scenario should be based on the technical needs and business impact of the application in an organization, by means of requirements related with the level of security that should be satisfied and the amount of resources available during development. As an example, for a high-quality scenario (e.g., home banking), one wants to select the **SAST** tool with the highest detection rate, even if it raises more false alarms than others, since any vulnerability that is left undetected may

have a high impact if it is successfully exploited. In this case, all resources that are required to check the warnings produced by the **SAST** tool and to fix the vulnerabilities are assumed to be available. On the other hand, for a medium-quality scenario (e.g., corporate site), one may want to choose the **SAST** tool with a high detection rate, but that does not raise too many false alarms, since the resources available to deal with those false alarms are not unlimited.

In our approach, we considered four criticality levels representing realistic scenarios. We adapted the names of the scenarios defined by Antunes et al. [159] to better represent their requirements, but maintaining their scope:

- 1) **Highest-quality**: every vulnerability missed may be a big problem due to the criticality of the application. For this scenario the goal is to select the **SAST** tool reporting the highest number of vulnerabilities even if reporting many false alarms.
- 2) **High-quality**: given that the criticality of applications is not the highest, a few vulnerabilities may be missed if that lowers the number of false alarms. For this scenario, the goal is to select the **SAST** tool reporting a high number of vulnerabilities but not too many false alarms.
- 3) **Medium-quality**: vulnerabilities may be missed at the cost of reducing the false alarms. For this scenario the goal is to select a **SAST** tool reporting few false alarms at the cost of skipping the detection of some vulnerabilities.
- 4) **Lowest-quality**: every false alarm is an important cause of concern due to tight budget restrictions. The goal for this scenario is to select the **SAST** tool reporting the lowest number of false alarms while still reporting vulnerabilities.

The definition and use of scenarios is very helpful for software developers and decision makers, because they can control the acceptable/expected outcomes of the **SAST** tools in the static analysis vulnerability detection process for each project that fits in a specific scenario. Moreover, identifying **SAST** tools that consistently under-perform can also be useful to avoid using them.

4.1.2. Benchmark Metrics

The benchmark measures are computed by analyzing the information reported by the **SAST** tools during the benchmark run. To compare the results and rank the benchmarked tools, we propose the use of metrics that are adequate to the vulnerability detection scenario. For each scenario, we propose one main metric to rank the **SAST** tools and a tiebreaker metric used only when there is a tie among tools (see Table 4.1), adapted from Antunes et al. [159]. In practice, the metrics depend on the vulnerability detection goals, which are related with the amount of available resources to fix the vulnerabilities. For example, in the *highest-quality scenario* the chosen metric is *recall*, which favors finding the highest number of vulnerabilities at any cost, even ignoring the precision of the results. Only in the case of a tie, the precision is used to rank first the tool that reports less false alarms.

A **SAST** tool can be viewed as a binary decision system for two-class problems, given that it has to classify the vulnerabilities and the non-vulnerabilities. For instance, the outputs of a **SAST** tool can be classified into four categories, which is the same for any other binary decision

Table 4.1.: Summary of Metrics by Scenario.

	Scenario	Metric	Tiebreaker
1	Highest-quality	Recall	Precision
2	High-quality	Informedness	Recall
3	Medium-quality	F-Measure	Recall
4	Low-quality	Markedness	Precision

system:

- Regarding *code that is not vulnerable* (Negative Instances (**N**)):
 - **False Positive (FP)**: the tool incorrectly determines that the code is vulnerable. The actual value is negative but the **SAST** tool predicted it as positive.
 - **True Negative (TN)**: the tool correctly determines that the code is not vulnerable. Both actual and predicted values are negative.
- For *code that is vulnerable* (Positive Instances (**P**)):
 - **False Negative (FN)**: the tool incorrectly determines that the code is not vulnerable. The actual value is positive but the **SAST** tool predicted it as negative.
 - **True Positive (TP)**: the tool correctly determines that the code is vulnerable. Both actual and predicted values are Positive.

The metrics used for evaluating the performance of the **SAST** tools and asserting its quality are based on a confusion matrix, which shows how the **SAST** tool (classifier) gets confused while predicting. The confusion matrix contains the number of correctly and incorrectly classified test cases for each class (**TP**, **FP**, **FN**, **TN**). Table 4.2 presents the confusion matrix for a two class problem. On the left side of the table, there are predicted values and on the top side there are the actual values. A perfect **SAST** tool should only present non-zero values in the confusion matrix main diagonal, as these correspond to correct classifications (**TPs** and **TNs**), while the remaining “cell” values represent miss-classified test cases (**FPs** and **FNs**) should have zero (or close to zero) values.

Based on the confusion matrix, we define two terms:

- 1) **Positive Instances (P)** as the number of vulnerabilities ($P = TP + FN$) in the source code of the web applications.
- 2) **Negative Instances (N)** as the number of non-vulnerabilities ($N = FP + TN$) in the source code of the web applications.

Table 4.2.: Confusion matrix for two class classification problem.

Predicted	Actual	
	<i>Positive Instances (P)</i>	<i>Negative Instances (N)</i>
	<i>Vulnerabilities</i>	<i>Non-vulnerabilities</i>
<i>Positive</i>	TP	FP
<i>Negative</i>	FN	TN

In the following, we describe the evaluation metrics and present some arguments about why the metrics portray the effectiveness of the **SAST** tools in each scenario. For instance, the metrics

focus on different parts of the confusion matrix, which allows them to define both absolute and relative metrics. Absolute metrics are based on the vulnerabilities and non-vulnerabilities in the workload and relative metrics are based on the vulnerabilities reported by **SASTs** tools). Note that, the *Recall* metric is based on all **P** instances, the *F-Measure* metric is based on all **P** instances and part of the **N** instances, the *Informedness* metric is based on all **P** and all **N** instances, the *Precision* metric is based on some **P** instances (i.e., **TPs**, the outcomes of the **SAST** tools), and the *Markedness* metric is based only on the outcomes of the **SAST** tools (i.e., part of **P** and part of **N** instances) [179]:

- **Recall.** The proportion of true vulnerabilities that are correctly identified as such, ranking first the **SAST** tools reporting the highest number of **TPs** required for the highest-quality scenario, and for the high-quality and medium-quality scenarios in case of a tie.

$$Recall = \frac{TP}{TP + FN} \quad (4.1)$$

- **Informedness.** How consistently a **SAST** tool predicts the outcome of both a **TP** and a **TN**, i.e., how informed a **SAST** tool is for the specified condition, versus chance. Every **TP** result increases the metric in the proportion $1/P$ and every **FP** result decreases the metric in the proportion $1/N$. Since the prevalence of **P** instances is less than the prevalence of **N** instances, the metric prioritizes **SAST** tools reporting more vulnerabilities and at the same time not too many **FPs**, which is the goal of the high-quality scenario.

$$Informedness = \frac{TP}{TP + FN} + \frac{TN}{FP + TN} - 1 = Recall + InverseRecall - 1 \quad (4.2)$$

- **Precision.** Proportion of the classified positive cases that are correctly classified. This metric is used only as tiebreaker. Thus, from a list of **SAST** tools reporting the same number of vulnerabilities, the best one is the **SAST** tool with highest Precision (i.e., less **FPs** reported).

$$Precision = \frac{TP}{TP + FP} \quad (4.3)$$

- **F-Measure.** The harmonic mean of recall and precision. In this metric, the **TPs** have twice the weight of the **FPs**. Thus, it is suitable for the medium-quality scenario where it is preferable to fix less than more vulnerabilities and at the same time to consume less resources checking **FPs**.

$$F-Measure = \frac{2 \times TP}{2 \times TP + FP + FN} \quad (4.4)$$

- **Markedness.** How consistently the **SAST** tool has the outcome as a marker, i.e. how marked a condition is for the specified **SAST** tool, versus chance. This metric sums the proportions of the positives and the negatives that are correctly identified as such. The Precision (1st part of the formula 4.5) focus on the **FPs** reported by the **SAST** tools and handles only part of the **N** instances. Therefore, based on the Precision, a **SAST** tool reporting no **FPs** is better than a **SAST** tool reporting all vulnerabilities but having at least one **FP**.

$$Markedness = \frac{TP}{TP + FP} + \frac{TN}{FN + TN} = Precision + InversePrecision \quad (4.5)$$

4.1.3. Building the Workload

The perfect workload is a large set of real applications of diverse sizes, developed according to typical industry practices and with all vulnerabilities identified [180]. However, such workload does not exist and creating it is a (probably unfeasible) task that would consume immense resources. To limit this problem, we propose a process based on the results of several **SAST** tools, combined with manual review to annotate the vulnerabilities and non-vulnerabilities in the real software used as workload.

The proposed process to build the workload is presented in Figure 4.2, and it involves three stages (illustrated by the gray boxes in the figure), which are discussed in the following subsections.

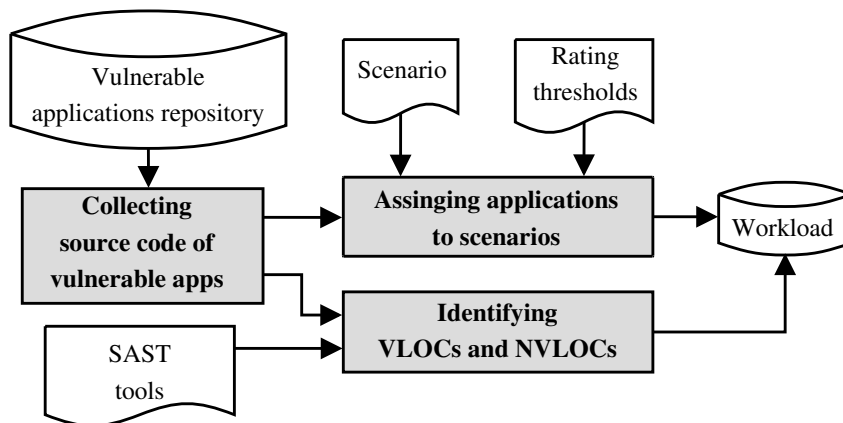


Figure 4.2.: Process to compose the workload.

4.1.3.1. Collecting the Source Code of Vulnerable Applications

The methodology to select a representative set of vulnerable applications to define the workload includes the following steps, represented in Figure 4.3:

- 1) Choosing applications in the benchmarking domain for which the source code is available. **SAST** tools require the access to the source of the application to detect vulnerabilities.
- 2) Choosing the classes of vulnerabilities that are relevant in the benchmark domain (e.g., **SQLi** and **XSS**).
- 3) Collecting all vulnerabilities of the chosen applications registered in their development repository or from vulnerability databases, e.g. WordPress Vulnerability Database (**WPVD**), **CVE**, and Electronic mailing list dedicated to issues about computer security (**Bugtraq**).
- 4) Selecting only vulnerabilities with a Proof of Concept (**PoC**), i.e., vulnerabilities for which a proof that they can be exploited exists (i.e., these are proven to be real exploitable vulnerabilities, which are much more interesting to attackers than supposedly existing vulnerabilities that may or may not be exploited).
- 5) Downloading the applications with the vulnerabilities with **PoC** from source code repositories.

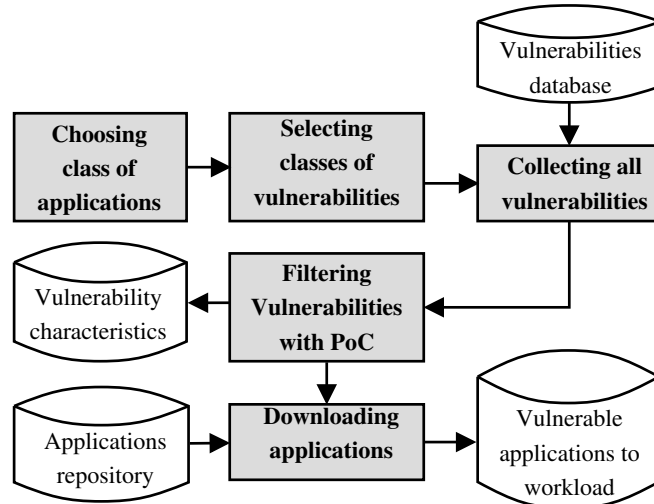


Figure 4.3.: Process for collecting vulnerable applications.

A major advantage of this methodology over existing benchmarks (like those from [NIST](#) and [OWASP](#)) is the representativeness of the vulnerabilities since they exist in real applications and are proven to be exploitable. In fact, we need workloads for which we have independent evidence of generalizability (at least some degree of) and ground truth.

4.1.3.2. Assigning Applications to Scenarios

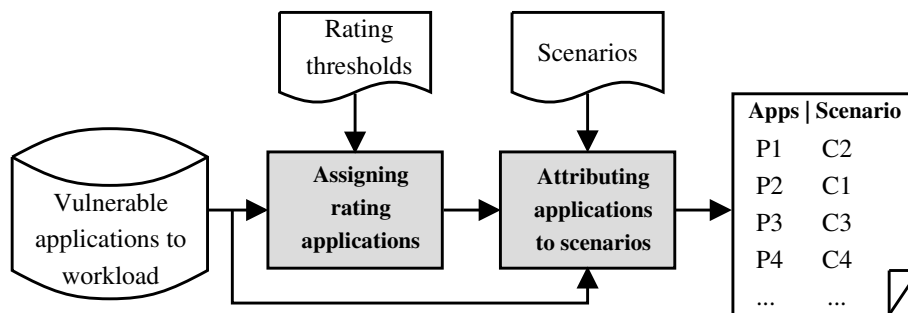


Figure 4.4.: Process for assigning applications to scenarios.

To compose the workload, we need to assign a representative set of vulnerable applications to each scenario. This process has two stages (see [Figure 4.4](#)):

- 1) **Assigning ratings to applications.** This stage is based on the approach proposed by Baggen et al. [181] for rating the maintainability of the source code of applications (from 1 to 5 stars in a discrete scale or from 0.5 to 5.5 stars in a continuous scale). The Baggen’s approach uses a standardized measurement model based on the ISO/IEC 9126 definition of maintainability and a small set of [Source Code Metrics \(SCMs\)](#) (e.g., Extended Cyclomatic Complexity Number (CCN2) [182]). These SCMs are used to measure the [Software Product Properties \(SPPs\)](#) (e.g., Unit Complexity) of the software. [Table 4.3](#) lists the SCMs used to measure the SPPs, including the level of measurement, and [Table 4.4](#) outlines the SPPs and

their relationship with the sub-characteristics of maintainability. This table also includes an example of assigning a rating to an application. The ratings of the sub-characteristics are obtained by averaging the ratings of the selected properties (marked with a “×”). The final rating is obtained by adding the average ratings and dividing by 4 (in the example: $(4.0 + 4.0 + 2.6 + 3.5)/4 = 3.5$ stars).

Table 4.3.: SCMs for Evaluating the SPPs.

SPP	SCM	Level	Description
Duplication	DLD	App.	Duplicated Line Density [183] [184]
Size	LLOC	Unit	Logical Lines of Code
Size	WMC	Class	Weighted Method Count [185]
Complexity	CCN2	Unit/Class	Extended Cyclomatic Complexity Number [182]
Coupling	CBO	Class	Coupling Between Objects [186]
Interfacing	NPARM	Unit	Number of parameters in functions and methods
Class Interface	CIS	Class	Number of non-private methods and properties [187]
Testing	NPATH	Unit	Number of execution paths

SPP - Software Product Property. SCM - Source Code Metric.

The Baggen’s approach is implemented in three major steps and requires as input: i) a large set of applications in the benchmarking domain; ii) a set of percentages of code to represent (i.e., range values for defining classes of code according to the values of the Source Code Metric (SCM)); and iii) a table of **rating thresholds** for each SCM. For i) we use the applications of our workload, and for ii) and iii) we adopt the same values proposed by Baggen et al., since they were successfully used in several works [181] [188]. In practice, the first step is the extraction of the values of the SCMs (there are many free tools for gathering the SCMs (e.g., PHPdepend [182])). Afterwards, using the values of all SCMs of all applications, we derive the ratings for each SCM of each application. Finally, we obtain the ratings of the applications by averaging the ratings of the sub-characteristics of maintainability, as described before. The approach for *Assigning ratings to applications* is detailed in Appendix A-Assigning Applications to Scenarios.

Table 4.4.: Mapping of Software Product Propertys (SPPs) to ISO/IEC Sub-Characteristics of Maintainability and an Example.

Sub-characteristic	Software Product Property (SPP)								Average Rating
	Duplication	Unit complexity	Unit size	Module coupling	Class complexity	Unit interfacing	Class interface size	Unit testing	
Rating example	5.0	4.0	3.0	4.0	3.0	3.0	2.0	3.0	
Analyzability	×	×	×						4.0
Changeability	×	×		×	×				4.0
Stability						×	×	×	2.6
Testability		×	×	×				×	3.5
Maintainability rating (average: ★★☆☆)									3.5

- 2) **Assigning applications to scenarios.** The second stage of the process is based on a simple schema for mapping the application ratings with the scenarios. Table 4.5 lists the mapping schema from ratings to scenarios. Since the ratings vary from 0.5 to 5.5 in ascendant quality order and the scenarios from 1 to 4 in descendant level of criticality, we used intervals of one unit for mapping the ratings into the scenarios, trying to respect Baggen’s approach, except for the less stringent (and probably less relevant) scenario, which accommodates the two lower ratings (code of less quality).

Table 4.5.: Mapping from Ratings to Scenarios.

Rating Range	Rating	#Scenario	Scenario
[4.5, 5.5]	5	1	Business-critical
[3.5, 4.5[4	2	Heightened-critical
[2.5, 3.5[3	3	Best-effort
[0.5, 2.5[1 and 2	4	Min-effort

4.1.3.3. Identifying VLOCs and NVLOCs

To evaluate **SAST** tools and compare them with each other requires to establish a ground truth of the actual vulnerabilities and non-vulnerabilities in the source code of the workload applications. Both have to be characterized in term of *location* (e.g., file name, line number in the source code, vulnerable sensitive sink, vulnerable variable). Therefore, it is fundamental to define precisely the vulnerabilities and non-vulnerabilities of the code in order to guarantee that, at the end of the assessment, there is no doubt as to where a tool reported vulnerabilities correctly and where it made false positive or false negative errors. Another problem that needs to be addressed relates with the fact that different **SAST** tools may report different types of vulnerabilities in different ways. Consider, for instance, a **SQLi** vulnerability where a **SQL** query that incorporates three different unsanitized user inputs is executed. Some **SAST** tools may report three different vulnerabilities (one for each input), while others may report only one (with three inputs) because the vulnerability occur in one **SS**. To provide accuracy in the computation of the metrics, we have to define how to count the number of vulnerabilities they report. To address these problems we propose the procedure discussed next.

We define a Vulnerable Line of Code (**VLOC**) as being a **LOC** with at least one vulnerability and a Non-Vulnerable Line of Code (**NVLOC**) as being a **LOC** without any vulnerability. A vulnerability may manifest in a restricted set of constructs of the programming language, named **SSs** (e.g., **XSS** in the PHP `echo` output function and **SQLi** in the `mysqli_query` PHP database function). A **SS** can be viewed as a location in the code that can be exploited if some maliciously crafted input is used as argument [189]. Although the number of Vulnerable Lines of Code (**VLOCs**) in an application is limited to the lines that include such constructs, the number of vulnerabilities can potentially be greater than the number of **SSs**, as one **SS** may have several vulnerabilities. For example, the PHP code `echo "$name $city;"`, may have two **XSS** vulnerabilities if the two user input variables are not sanitized between their input and output in the **SS**. Moreover, it is common to have a single **LOC** with several **SSs**. For example, consider the following line of PHP code that dynamically generate attributes and content of **HTML** tags in

the form: “<tag attribute1=<?php echo \$value1 ?>” attribute2=<?php echo \$value2 ?>><?php echo \$content ?></tag>”. This LOC has three SSs (echo PHP function) using different variables, and each one may be vulnerable depending on the context: in this case HTML tags, attributes and content. It is also common to find LOCs with several SSs sharing one or more variables, and a single SS using the same variable more than once as arguments. In these cases, it is necessary to know the position of the SS inside the LOC and also the position of the vulnerable variable.

It is important to mention that, although SAST tools report alerts for a particular type of vulnerability, it can not be said that each TP corresponds to a single vulnerability. In fact, an alert may correspond to more than one vulnerability if the SS uses more than one LOC. The results presented in this work refer to the vulnerable SSs and not the total number of vulnerabilities, as the SAST tools report one occurrence for each vulnerable SS. In the next paragraphs, we discuss how to characterize the VLOCs and Non-Vulnerable Lines of Code (NVLOCs) and present the method to obtain them.

The **list of Vulnerable Lines of Code (VLOCs)** in the workload is composed by two disjoint sets. The first set of VLOCs are the vulnerabilities (i.e., the LOCs where the vulnerabilities are located) resulting from the process of collecting the source code of vulnerable applications (as discussed in Section 4.1.3.1), specifically, exploitable vulnerabilities registered in public vulnerability databases. However, the number of these vulnerabilities is typically very low, probably lower than the real number, breaking the ground truth of the workload (i.e., the potential vulnerable LOC are not characterized in term of VLOCs and NVLOCs). Our approach to find additional VLOCs in the workload is based on running several SAST tools and do a manual review by security experts to confirm the results. Thus, to obtain the second set of VLOCs, we run SAST tools to scan for vulnerabilities in the selected applications; then, the outputs are combined and each candidate vulnerability is manually reviewed to determinate if it is a TP (i.e., vulnerability) or a FP (i.e., non-vulnerability). Thus, the first set of VLOCs together with the second set becomes the list of VLOCs (i.e., P) and all FPs becomes part of the list of NVLOCs. Obviously, this approach is limited in term of completeness, as even if we use many SAST tools to find vulnerabilities in the workload, some vulnerabilities may not be detected due the limitations of the SAST tools. Therefore, if available, other approaches for detecting vulnerable lines of code may be integrated in the benchmarking process.

The **list of Non-Vulnerable Lines of Code (NVLOCs)** in the workload is also composed by two disjoint sets. The first is composed by the FPs reported by the SAST tools used in the process of characterizing the VLOCs (which were manually confirmed). Consequently, if the SAST tools used in the process report few FPs, the size of the set will be small, so the values of the metrics that depend on the number of NVLOC (e.g., informedness) would not be representative if only those were considered. A naive way to identify more NVLOCs would be to calculate the difference between the total number of LOCs of the application that compose the workload and the number of NVLOCs. However, this would result in an extreme unbalance between VLOCs and NVLOCs, because of an unrealistic very high number of NVLOCs. In this case, FPs would have a very small (or negligible) effect on the metrics based on the NVLOCs. For example, the results for the informedness metric would become very similar (slightly lower) to the results for the recall metric, thus losing its usefulness. To overcome this problem, we

propose to consider as **NVLOCs** only the **LOCs** that constitute a **SS** with at least one variable, but for which no vulnerability is known or has been detected. In fact, **SS** function calls without any variable are not possible to be vulnerable, thus they are not considered as **NVLOCs**.

The procedure for **obtaining VLOCs and NVLOCs** is, in practice, a seven-step process as follows:

- 1) Collect **VLOCs** with **PoC** of the web applications in the workload from public vulnerability's databases.
- 2) Identify the set of **SAST** tools to be used for defining the list of **VLOCs** and **NVLOCs**. This includes the definition of the configuration settings for the selected **SAST** tools according to the classes of vulnerabilities to be searched.
- 3) Detect vulnerabilities by running the **SAST** tools on the web applications of the workload. From this step results a list of candidate **VLOCs**.
- 4) Manually verify the vulnerabilities reported by the **SAST** tools and classify them either as **VLOCs** or **NVLOCs**.
- 5) Create the list of **VLOCs** by joining the initial set of vulnerabilities with the **PoC** that resulted from the process of collecting the source code of vulnerable web applications and the **VLOCs** from Step 4).
- 6) Create the set of **NVLOCs** by joining all distinct **FPs** reported by the **SAST** tools, which will compose the first set of **NVLOCs** in the workload. The second set is composed by the **LOCs** where a **SS** function is called, having at least one variable, but excluding those that were labeled as **VLOCs** in the previous step.
- 7) Characterize the set of **VLOCs**, including information of the vulnerable file, the **EPs** and **SS**, the **LOC** number, the vulnerable variable and the class of vulnerability.

We are aware that the process for extracting **VLOCs** and **NVLOCs** may leave some vulnerabilities undetected. Consequently, an issue may occur during the execution of the benchmark if one or more **SAST** tools report previously unknown vulnerabilities. This requires a manual review to classify such findings as **TPs** or **FPs**. This allows updating the list of **VLOCs** and **NVLOCs**, but also changes the values of the metrics/ranking of **SAST** tools previously benchmarked, which may also need to be updated. Although a best effort approach, the usage of several **SAST** tools in the **LOCs** characterization process would minimize this problem, therefore reducing the probability of a **SAST** tool to detect unknown vulnerabilities.

4.1.4. Procedure and Rules

The benchmarking procedure is a well-defined set of steps and rules that must be followed to implement and run the benchmark (see Figure 4.5):

- 1) **Preparation:** identifying the **SAST** tools to be benchmarked. Different tools are executed in different ways, as they have diverse features, configurations and user interfaces, thus, whenever possible, the tools must be configured according to the characteristics of applications in the benchmark domain.

- 2) **Execution:** running the **SAST** tools under benchmarking to detect vulnerabilities in the workload.
- 3) **Normalization of reports:** as each tool delivers the results in a specific format, they must be normalized and merged into a single report with a common format, including the following information: the **LOCs** reported as vulnerable, the files where they were found, the vulnerability class, and the web application where they were discovered.
- 4) **Vulnerability verification:** analysis of the results of the **SAST** tools by applying three verifications:
 - i) The vulnerabilities reported by the **SAST** tools belonging to the list of **VLOCs** (i.e., **TP**) are automatically verified;
 - ii) The vulnerabilities reported by the **SAST** tools belonging to the set of **FPs** in the list of **NVLOCs** are also automatically verified;
 - iii) The vulnerabilities reported by the **SAST** tools that belong to the set of **SSs** of the list of **NVLOCs** require a manual verification to confirm their veracity. The lists of **VLOCs** and **NVLOCs** should be updated according to the results of the manual review:
 - a) *if the vulnerability reported is a **TP**:* the entry from the set of **SSs** of the list of **NVLOCs** is removed and added to the list of **VLOCs**.
 - b) *if the vulnerability reported is a **FP**:* in the list of **NVLOCs** move the entry from the set of **SSs** to the set of **FPs**.
- 5) **Metrics calculation and ranking:** based on the outputs of the **SAST** tools and their verification (previous step), the benchmark metrics are calculated automatically. Afterwards, **SAST** tools are ranked according to the metrics recommend for each scenario (see Table 4.1 for more details).

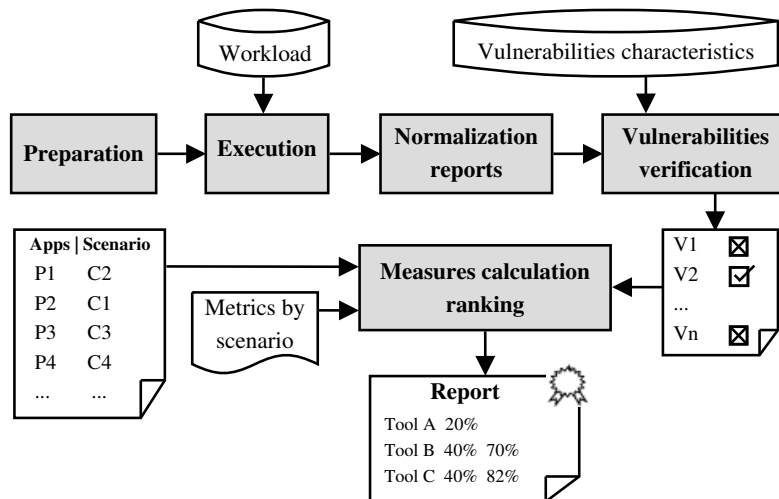


Figure 4.5.: Benchmarking procedure.

For intanciating the benchmark approach, we need define its scope, create the workload and follow the procedures and rules during the execution of the benchmark. The following section describes an instance of the benchmark approach for WordPress plugins and **SQLi** and **XSS** vulnerabilities for evaluating and ranking five free **SAST** tools.

4.2. Benchmark Instantiation

The benchmarking approach proposed in the previous section intends to be generic, meaning that it may be applied to any type of application and class of vulnerability for the evaluation of any set of [SAST](#) tools. This section presents a concrete benchmark developed following the proposed approach. The workload are WordPress plugins and the target vulnerabilities are [SQLi](#) and [XSS](#), which are two of the most common web application security vulnerabilities [15] and also two of the most widely exploited [50]. Statistics say that 98% of WordPress vulnerabilities are related to plugins and most root cause of WordPress hacking incidents is outdated vulnerable plugins [190].

WordPress is extensible by means of [PHP](#)-based resources such as plugins that allow the addition of new features, templates, functions, etc. They are so common that the WordPress Plugins Dataset ([WPD](#)) contains around 58,931 plugins (on July 2021) with over 1.5 billion downloads [191]. Moreover, there are dozens of thousands of other third-party plugins and themes available for free or for purchase. As an example of one of these plugins, there is the [WooCommerce](#), which is one of the most successful WordPress plugins. It offers entrepreneurs the ability to easily open an online store and sell their products to anyone. [WooCommerce](#) powers over 28% of all online stores. For this plugin, there are over 1,260 [WooCommerce](#) themes on [ThemeForest.net](#) and there are over 980 plugins for [WooCommerce](#) on [WordPress.org](#) [192]. In fact, plugins are responsible for lots of vulnerabilities and, since a single plugin may be used in thousands of websites, they are an attractive target for hackers. For example, the plugin [Contact Form 7](#)¹ has more than 5 million active installations and only in the [WPVD](#)² there are registered several vulnerabilities including [SQLi](#), [XSS](#), [RCE](#), [Privilege Escalation \(PE\)](#) among others.

The next subsections describe the composition of the workload, developed following the process to create the workload presented in Section 4.1.3, based on vulnerable WordPress plugins registered in the [WPVD](#).

4.2.1. Collecting the Source Code of Vulnerable Applications

We used the online [WPVD](#) [193] to collect WordPress plugins with [SQLi](#) and [XSS](#) vulnerabilities, including [PoC](#) and more details, like the [CVE](#) identifier. For collecting the plugins we proceed as follows: first (on 16-10-2015) we collected the list of all the vulnerabilities in the repository. The list includes a slug, date and title for each vulnerability. Then, from 2,779 vulnerabilities collect we select only 143 by applying a filter to select the ones that include in the title: “[SQLi](#)”, “[SQL Injection](#)” “[XSS](#)” “[Cross-Site Scripting](#)” [XSS](#). finally, from this list we created unique list of 134 plugins by removing vulnerabilities without [PoC](#) and considering a plugin name once.

The result of applying the step “[Collecting the Source Code of Vulnerable Applications](#)”, as proposed in Section 4.1.3.1, was a list of 134 WordPress plugins with 152 [SQLi](#) and 67 [XSS](#) vulnerabilities registered. Forty-two of these plugins contain both classes of vulnerabilities,

¹<http://contactform7.com>

²<http://wpvulndb.com/search?text=contact+form+7>

while 79 contain only [SQLi](#) and 13 only [XSS](#). To have an idea of their relevance, overall these plugins have been downloaded over 77 million times and they are used in business, e-commerce, monetization, social networking (Google, Facebook, Youtube, etc.), photo and video gallery, registration, admin, advertising, email, bookings, reservations, events management, newsletter, e-learning and document manager. The list of plugins, including rating information, distribution per scenario, vulnerabilities and further results are given in [Appendix B-List of WordPress plugins](#). This data is also available online [194]. It is important to emphasize that, using the workload that we created, researchers can evaluate other [SAST](#) tools with little effort.

Some of the plugins obtained were developed in PHP using Procedure Oriented Programming ([POP](#)) and others using [OOP](#). Notice that, in PHP the use of classes does not imply an object-oriented design. Thus, it is frequent to find procedural code using objects and [OOP](#) code mixed with procedural code. We considered the plugins containing code with definition of classes as [OOP](#) plugins. Overall, we have 31 [POP](#) plugins and 103 [OOP](#) plugins, as shown in [Table 4.6](#). The workload of plugins contains a total of 466,164 Logical Lines of Code ([LLOC](#)), where 39.5% are [POP](#), 47.8% [OOP](#), and 12.7% a mix of both. The number of [LOC](#) is 1,023,081 where 32% are [POP](#), 57% [OOP](#), and 11% a mix of both. The table also shows the number of vulnerabilities with [PoC](#) by class of vulnerability.

Table 4.6.: Plugin background information by type of code.

Code Type	Plug	Files				LLOC				WPVD		Code				
		Total	Min	Max	Avg	OOP	POP	Total	Min	Max	Avg	SQLi	XSS	CL	F	M
OOP	103	297	1	40	9	222695	218414	441109	191	11077	1750	84	13	0	1089	0
POP	31	4693	1	559	45	0	25055	25055	188	89564	9932	0	0	2518	7370	21189
Total	134	4990				222695	243469	466164				84	13	2518	8459	21189

Plug-Plugins CL-Total classes, F-Total functions, M-Total methods.

4.2.2. Assigning Applications to Scenarios

For gathering the measures of the [SCMs](#) to evaluate the [SPPs](#) listed in [Table 4.3](#), we used three tools: [PHPdepend v2.5.0³](#) [182] for gathering the [LLOC](#), the Weighted Method Count ([WMC](#)), the [CCN2](#), the Coupling Between Objects ([CBO](#)), the Number of non-private methods and properties ([CIS](#)), and the Number of execution paths ([NPATH](#)) metrics; [SonarQube v5.2⁴](#) [195] for the Duplicated Line Density ([DLD](#)) metric; and [PHPMD v2.6.0⁵](#) for the Number of parameters in functions and methods ([NPARM](#)) metric.

The results of applying the step “[Assigning Applications to Scenarios](#)” (as proposed in [Section 4.1.3.2](#)) are presented in [Table 4.7](#), which shows the number of plugins that compose the workload, distributed over four scenarios. Scenario 1 (highest-quality) has a lower number of plugins compared with scenario 2 (high-quality) and scenario 3 (medium quality).

³<https://pdepend.org>

⁴<http://www.sonarqube.org>

⁵<https://phpmd.org/>

Table 4.7.: Plugin background information by scenario and type of code.

	Scenario	OOP	POP	Total	%Total	Files	LLOC	%LLOC
1	Highest-quality	10	2	12	8.9	352	19,542	4.2
2	High-quality	39	17	56	41.8	1,687	122,835	26.4
3	Medium-quality	40	11	51	38.1	2,208	211,297	45.3
4	Low-quality	14	1	15	11.2	728	112,490	24.1
	Total	103	31	134	100.0	4,975	466,164	100.0

4.2.3. Identifying Vulnerabilities and Non-vulnerabilities

The first part of the list of **VLOCs** was given by the information collected from the WordPress Vulnerability Database (**WPVD**). To obtain the second part, we ran two free **SAST** tools, **RIPS** [51] and **phpSAFE**[109], to scan for vulnerabilities in the workload. The **SAST** tools were configured by default for PHP entry points, **SS** and **FPs** sanitization functions (e.g., `htmlentities`, `mysql_real_escape_string`). The results were manually reviewed as defined in our design approach (Section 4.1.3).

The list of **NVLOCs** considered is the combination of the **FPs** reported by the tools with the list of **LOCs** that have at least one **SS** outputting at least one variable. For this, we developed a PHP script for gathering all **SS** function calls of the source code files based on their **AST**. From this list, we removed those already labeled as **VLOCs**. The script was executed individually for each file. A manual check of random samples has been performed to increase the trust on the accuracy of the **NVLOCs** identified.

Table 4.8 presents the results obtained. Overall, 6,725 (**FP**: 1,294 + **TP**: 5,431) **LOCs** were extracted from the outputs of the **SAST** tools and manually reviewed (80.8% of **TPs** and 19.2% of **FPs**). The table depicts, for each tool, the number of **TPs** and **FPs**, followed by the number of vulnerabilities in the **WPVD** (column **VD**) [193]. The column **Total FP** is the union of the **FPs** of the tools, and **NV** shows the **NVLOCs** obtained in the previous step (**VLOC** characterization). The two last columns show the number of **P** and **N** (combination of the **FPs** with the **NV**). These columns are used for calculating the evaluation metrics and should be updated during the execution of the *benchmarking procedure* if a **SAST** tool under testing reports a previously unknown vulnerability.

An important aspect regarding the **VLOCs** is that the number of vulnerabilities reported in the **WPVD** (97, as shown in Table 4.8) is far from the reality (5,431, as in Table 4.8). In fact, we were able to detect a much larger number of true vulnerabilities using the **SAST** tools. This emphasizes the capability and relevance of static analysis to detect vulnerabilities.

4.3. Experimental Evaluation

The main goal of this experimental evaluation is to demonstrate the benchmark proposed in the previous section, validate the benchmarking process and, at the same time, confirm/deny the following hypothesis:

Table 4.8.: Distribution of Vulnerabilities and Non-Vulnerabilities by Scenarios and tools/VD.

Scenario	phpSAFE		RIPS		VD	Total	NV	Total		
	TP	FP	TP	FP		FP		P	N	
SQLi	1	29	5	0	0	17	5	84	41	89
	2	274	58	43	2	35	60	1,068	308	1,128
	3	99	50	153	113	22	163	2,053	251	2,216
	4	36	32	1	0	10	32	1,105	46	1,137
	Total	438	145	197	115	84	260	4,310	646	4,570
XSS	1	96	16	113	29	3	43	947	168	990
	2	1,149	76	887	188	1	223	5,673	1,767	5,896
	3	951	264	1,775	487	4	652	9,370	2,315	10,022
	4	244	33	369	89	5	116	3,481	535	3,597
	Total	2,440	389	3,144	793	13	1,034	19,471	4,785	20,505
Total	2,878	534	3,341	908	97	1,294	23,781	5,431	25,075	

H_1 : The best **SAST** tool is the same across different scenarios.

H_2 : The best **SAST** tool is the same across different classes of vulnerabilities.

We focus on free **SAST** tools as both occasional developers and professional software houses wanting to speed up the development process and reduce cost tend to use free tools as much as possible. Furthermore, such tools are easily available for research and results can be published without infringing licensing agreements. In practice, we evaluated the following tools: RIPS v0.55 [51], Pixy v3.03 [81], phpSAFE [109], WAP v2.0.1 [103], and WeVerca v20150804 [100]. RIPS and Pixy are the two most referenced PHP **SAST** tools in the literature, but they are not ready for **OOP** analysis. Pixy performs tainted analysis and alias analysis, but has not been updated since 2007, and RIPS has only been developed as open source until 2014. Then, ‘‘RIPS Technologies GmbH⁶’’ released a commercial version of RIPS able to fully analyze **OOP** code [196]. WAP, phpSAFE, and WeVerca are more recent tools under active development, and they are prepared for **OOP** code. In terms of configuration, phpSAFE, RIPS, WAP and Pixy are configured by default for PHP **EPs**, **SSs** and sanitization functions (e.g., `htmlentities`, `mysql_real_escape_string`). WeVerca does not allow configuration and includes, out of the box, a programmed list of **EPs**, **SSs** and Sanitization Functions (**SFs**).

The application of the ‘‘Procedures and Rules’’ to run the benchmark is described in detail in Appendix C-Benchmarking Procedure and Rules. In the following subsections, we present and discuss the results. We first present, in Section 4.3.1, details about the execution of the benchmark and discuss the results. In Section 4.3.2 we compare our results with the results of using **SAMATE** and **BSA** metrics. Section 4.3.3 discuss the limitations and benchmark properties.

4.3.1. Ranking the SAST Tools

We ran the benchmark for all the **SAST** tools searching for **XSS** and **SQLi** vulnerabilities in the workload plugins. Overall, WAP was able to analyze all plugins, but seven of them only partially. Pixy analyzed partially 103 plugins (i.e., fails in 1473 files) and WeVerca was not able to analyze 20 source files of 14 plugins. phpSAFE was unable to fully analyze 18 plugins (130 files), taking

⁶<https://www.ripstech.com>

Table 4.9.: Ranking of Tools by Scenario: [SQLi](#).

Scenario/Tool	TP	FP	FN	TN	Plugins	Main Metric	Tiebreaker Metric
<i>Highest-quality</i>						<i>Recall</i>	<i>Precision</i>
WAP	49	4	26	83	7	0.653	0.925
phpSAFE	29	5	46	82	5	0.387	0.853
WeVerca	0	0	75	87	0	0.000	-
RIPS	0	0	75	87	0	0.000	-
Pixy	0	0	75	87	0	0.000	-
<i>High-quality</i>						<i>Informedness</i>	<i>Recall</i>
phpSAFE	274	58	72	1,057	30	0.740	0.792
WAP	44	4	302	1,111	12	0.124	0.127
RIPS	43	2	303	1,113	8	0.123	0.124
WeVerca	18	1	328	1,114	6	0.051	0.052
Pixy	16	0	330	1,115	7	0.046	0.046
<i>Medium-quality</i>						<i>F-Measure</i>	<i>Recall</i>
RIPS	153	113	114	2,101	6	0.574	0.573
phpSAFE	99	50	168	2,164	15	0.476	0.371
WAP	72	0	195	2,214	11	0.425	0.270
Pixy	54	13	213	2,201	4	0.323	0.202
WeVerca	21	34	246	2,180	3	0.130	0.079
<i>Low-quality</i>						<i>Markedness</i>	<i>Precision</i>
WAP	5	0	45	1,137	2	0.962	1.000
RIPS	1	0	49	1,137	1	0.959	1.000
phpSAFE	36	32	14	1,105	7	0.517	0.529
WeVerca	0	0	50	1,137	0	-	-
Pixy	0	0	50	1,137	0	-	-

a very long time on those plugins without returning any results. RIPS outputted the message "Code is object-oriented. This is not supported yet and can lead to false negatives" for 76 plugins (2179 files). In practice, the tools could not fully analyze some plugins/files, reporting runtime errors or taking a very long time without any results. This results from limitations of the [SAST](#) tools used, potentially due to the size/complexity of some files.

The results by scenario for [SQLi](#) and [XSS](#) vulnerabilities are listed in Table 4.9 and Table 4.10, respectively. The columns **TP**, **FP**, **FN**, and **TN** show the confusion matrix for the corresponding [SAST](#) tool. The data in the tables are firstly ordered by the main metric (*Metric*). The *Plugins* column shows the number of plugins where the [SAST](#) tools found vulnerabilities.

Focusing on [SQLi](#) (Table 4.9), the tool chosen for each scenario was: **WAP** for the *highest-quality* scenario; **phpSAFE** for the *high-quality*; **RIPS** for the *medium-quality scenario*, despite having detected vulnerabilities in just a few plugins (6); and **WAP** for the *low-quality scenario*, with few vulnerabilities found (5) and zero **FPs**. As for [XSS](#) vulnerabilities (Table 4.10), **RIPS** was the best [SAST](#) tool for the *highest-quality* and *the medium-quality scenarios*; **phpSAFE** was the best [SAST](#) tool for the *high-quality scenario*; and **WAP** was the best [SAST](#) tool for the *low-quality scenario*. Unlike for [SQLi](#), all tools found [XSS](#) vulnerabilities in all scenarios.

Some key observations are worth being mentioned regarding the results for [SQLi](#) (Table 4.10):

- *Highest-quality scenario (recall metric)*: **WAP** comes 1st with 49 vulnerabilities found and 4

FPs. phpSAFE comes in 2nd with 29 vulnerabilities. The remaining tools did not find any vulnerabilities. The reason is that RIPS and Pixy do not fully analyze OOP and in this scenario 10 out 12 plugins are OOP (recall that this is the scenario with less plugins).

- *High-quality scenario (informedness metric):* phpSAFE ranked 1st, detecting both the highest number of **TPs** (274) and **FPs** (58). WAP comes in 2nd place with about 1/6 of **TPs** (44) and few **FPs** (4). RIPS comes in 3rd with similar results. WeVerca and Pixy have similar results but only found about half of the vulnerabilities found by RIPS.
- *Medium-quality scenario (F-Measure metric):* RIPS comes in 1st place, with the highest number of **TPs** (153), despite having detected vulnerabilities in just a few plugins (6) and much more **FPs** (113) than the other tools. This occurred because, in *F-Measure*, **TPs** are prevalent over **FPs**. phpSAFE comes in 2nd with 2/3 of the vulnerabilities of RIPS and WAP in 3rd with 70 **TPs** and no **FPs**.
- *Low-quality scenario (markedness metric):* the tool that ranks first is WAP with few vulnerabilities found (5) and zero **FPs**. RIPS comes in 2nd with only one vulnerability found, and also zero **FPs**. phpSAFE comes in 3rd, despite having detected many **TPs** (36). However, it also reported many **FPs** (32). The tools WeVerca and Pixy did not report any vulnerabilities.

Focusing on **XSS** (Table 4.9), unlike for **SQLi**, all tools found vulnerabilities in all scenarios. Some key observations are:

Table 4.10.: Ranking of Tools by Scenario: **XSS**

Scenario/Tool	TP	FP	FN	TN	Plugins	Main Metric	Tiebreaker Metric
						<i>Recall</i>	<i>Precision</i>
<i>Highest-quality</i>							
RIPS	113	29	55	961	10	0.673	0.925
phpSAFE	102	18	66	972	8	0.607	0.853
Pixy	69	14	99	976	7	0.411	-
WeVerca	44	5	124	985	7	0.262	-
WAP	23	6	145	984	3	0.137	-
						<i>Informedness</i>	<i>Recall</i>
<i>High-quality</i>							
phpSAFE	1164	90	678	5,735	46	0.617	0.792
RIPS	1013	194	829	5,631	46	0.517	0.127
WeVerca	436	50	1,406	5,775	25	0.228	0.124
Pixy	453	148	1,389	5,677	28	0.221	0.052
WAP	219	55	1,623	5,770	18	0.110	0.046
						<i>F-Measure</i>	<i>Recall</i>
<i>Medium-quality</i>							
RIPS	1,812	490	577	9,479	43	0.773	0.573
phpSAFE	970	267	1,419	9,702	41	0.535	0.371
Pixy	717	56	1,672	9,913	23	0.454	0.270
WeVerca	621	21	1,768	9,948	19	0.410	0.202
WAP	344	13	2,045	9,956	18	0.251	0.079
						<i>Markedness</i>	<i>Precision</i>
<i>Low-quality</i>							
WAP	62	3	483	3,591	6	0.835	1.000
phpSAFE	244	33	301	3,561	10	0.803	1.000
WeVerca	73	8	472	3,586	7	0.785	0.529
RIPS	377	91	168	3,503	10	0.760	-
Pixy	51	7	494	3,587	9	0.758	-

- *Highest-quality scenario*: RIPS is the best tool with both the highest number of vulnerabilities detected (113) and FPs (29). phpSAFE comes in 2nd with similar TPs, but with half of the FPs.
- *High-quality scenario*: phpSAFE is the best with the highest number of vulnerabilities (1164) and with half of the FPs of RIPS which appears in the 2nd place. On the other hand, RIPS is the 1st for the medium-quality scenario, both with the highest number of TPs and FPs. phpSAFE comes in 2nd with both half of TPs and FPs of RIPS.
- *Medium-quality scenario*: RIPS comes in 1st both with the highest number of TPs and FP. phpSAFE comes in 2th both with about half of the number of TPs and FP reported by RIPS.
- *Low-quality scenario*: like for SQLi, WAP comes in 1st with several TPs (62) and only 3 FPs. phpSAFE comes in second with about 4 times more TPs, but also with more than 10 times of FPs. RIPS ranks in 4th place with 5 times more TPs and 30 times more FPs.

In general, the results show that the best solution for vulnerability detection depends on the chosen scenario and on the class of vulnerabilities. Therefore, *hypotheses H_1 (the best SAT is the same across different scenarios) and H_2 (the best SAT is the same across different classes of vulnerabilities) are both false*. In fact, the detection capabilities of the SAST tools are not uniform across the two classes of vulnerabilities, nor across scenarios even if considering the same class of vulnerabilities. A relevant observation is that, in almost all cases, the SAST tools analyzed are better at detecting XSS than SQLi.

We also verified whether or not the metrics we used were the best to rank the SAST tools in each scenario. To confirm this, we simulated, in Section 4.3.2, the ranking procedure using the other metrics and compared these results with those that we have in Tables 4.9 and 4.10. For example, we concluded that without scenarios and the respective metrics, both strengths and limitations of the tools may be masked.

4.3.2. Results for SAMATE and BSA Metrics

In this section, we compare our results with the results of using SAMATE [167] and BSA [117] evaluation metrics, in order to show the capabilities and limitations of the different metrics for ranking SAST tools. First, we present the results on ranking the five SAST tools using the SAMATE and BSA metrics, and then we compare our rankings with the ones from SAMATE and BSA. Figures 4.6 to 4.8 plot the TPR versus FPR to provide a visual indication of the results of the SAST tools, showing how well each tool finds TPs and avoids FPs (see Section 2.6 for more details on the two benchmarks).

Unlike our approach, SAMATE and BSA are not organized in scenarios. These benchmarks use their metrics for all applications in the workload and regardless the resource available for vulnerability detection. To make the results comparable we calculated those metrics for the plugins we have in each scenario.

Using SAMATE and BSA evaluation metrics. Except for the DR, the SAMATE metrics coincide with some of our metrics. The results for those where presented before in the tables 4.9 and 4.10. As the remaining metrics of SAMATE (*Recall*, *F-Score* and *Precision*) are not

used for explicitly ranking the tools, we do not include that analysis here (i.e., the *F-Score* for the scenario highest-quality, the *Precision* and the *Recall* for the scenario medium-quality, etc.). However, they can be calculated with the data in the referred tables. Using the data from tables 4.9 and 4.10, we computed the values of the *BSA* metrics, as shown in Table 4.11 for *SQLi* and Table 4.12 for *XSS* (the tools are sorted using the *BAS* metric). Note that, since the *BAS* metric is based on the *Informedness*, the ranking of the tools for the high-quality scenario is the same of our benchmark.

For all scenarios and classes of vulnerabilities, the values of the *TPR* are at least 10 times higher than the values of the *FPR*. This shows the importance of identifying the *NVLOCs* in production software, to allow characterizing the strengths and limitations of the tools. In fact, the tools are not reporting *FPs* in many of the *locations* where the sensitive sinks are, which are the *locations* where a tool may find a vulnerability.

Table 4.11.: Ranking of Tools by Scenario and *BAS* Metric: *SQLi*.

Highest-quality				High-quality			
<i>Tools</i>	<i>TPR</i>	<i>FPR</i>	<i>BAS</i>	<i>Tools</i>	<i>TPR</i>	<i>FPR</i>	<i>BAS</i>
WAP	65.3	4.6	60.7	phpSAFE	79.2	5.2	74.0
phpSAFE	38.7	5.7	32.9	WAP	12.7	0.4	12.4
WeVerca	0.0	0.0	0.0	RIPS	12.4	0.2	12.2
RIPS	0.0	0.0	0.0	WeVerca	5.2	0.1	5.1
Pixy	0.0	0.0	0.0	Pixy	4.6	0.0	4.6
Medium-quality				Low-quality			
RIPS	57.3	5.1	52.2	phpSAFE	72.0	2.8	69.2
phpSAFE	37.1	2.3	34.8	WAP	10.0	0.0	10.0
WAP	27	0.0	27	RIPS	2.0	0.0	2.0
Pixy	20.2	0.6	19.6	WeVerca	0.0	0.0	0.0
WeVerca	7.9	1.5	6.3	Pixy	0.0	0.0	0.0

Table 4.12.: Ranking of Tools by Scenario and *BAS* Metric: *XSS*

Highest-quality				High-quality			
<i>Tools</i>	<i>TPR</i>	<i>FPR</i>	<i>BAS</i>	<i>Tools</i>	<i>TPR</i>	<i>FPR</i>	<i>BAS</i>
RIPS	67.3	2.9	64.3	phpSAFE	63.2	1.5	61.6
phpSAFE	60.7	1.8	58.9	RIPS	55	3.3	51.7
Pixy	41.1	1.4	39.7	WeVerca	23.7	0.9	22.8
WeVerca	26.2	0.5	25.7	Pixy	24.6	2.5	22.1
WAP	13.7	0.6	13.1	WAP	11.9	0.9	10.9
Medium-quality				Low-quality			
RIPS	75.8	4.9	70.9	RIPS	69.2	2.5	66.6
phpSAFE	40.6	2.7	37.9	phpSAFE	44.8	0.9	43.9
Pixy	30.0	0.6	29.5	WeVerca	13.4	0.2	13.2
WeVerca	26.0	0.2	25.8	WAP	11.4	0.1	11.3
WAP	14.4	0.1	14.3	Pixy	9.4	0.2	9.2

Figure 4.6 and Figure 4.7 present the chart plots (similar to the ones provided by the *OWASP BSA*) showing the results of the tools by scenario for *SQLi* and *XSS*, respectively. The order of the items in the captions stands for the order of the tools ranked using the *BAS* metric. The charts include the average of all tools. As shown, all tools score above the diagonal line, i.e.,

TPR is greater than FPR. For example, for [SQLi](#) and highest-quality scenario, WAP comes first with the best TPR but not with the best FPR. The tool is largely better than the average of all tools.

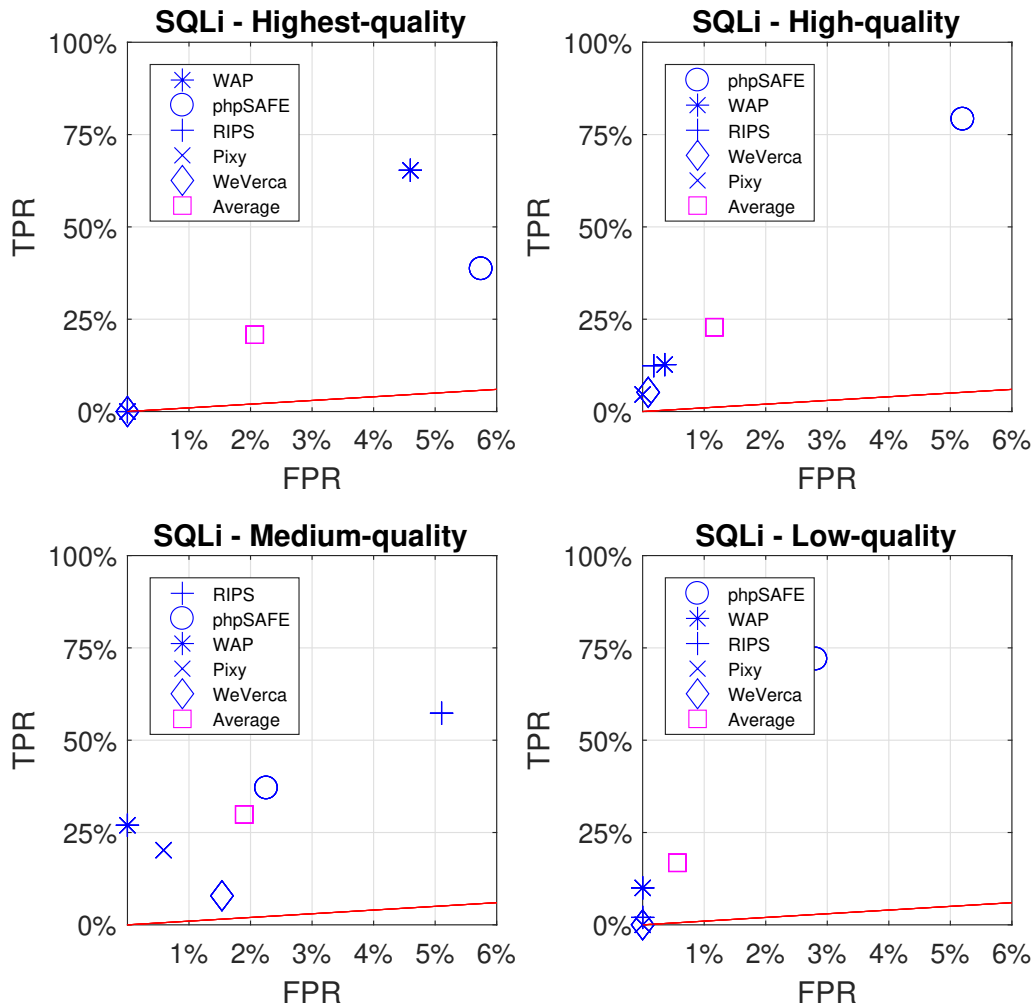


Figure 4.6.: Benchmark [SQLi](#) comparison by scenario.

Table 4.13 details the values of our main metrics for [SQLi](#) and [XSS](#) vulnerabilities considering the inexistence of scenarios (without assigning scenarios to the plugins). We observe that, depending on the class of vulnerability, the same tool comes first for almost all the metrics: [phpSAFE](#) for [SQLi](#) and [RIPS](#) for [XSS](#). Table 4.14 presents the results using the [BSA](#) metrics not considering the scenarios, and Figure 4.8 shows the respective graphs. The results are similar to the results of using our main metrics, thus [phpSAFE](#) is better for [SQLi](#) and [RIPS](#) for [XSS](#). We verified that both tools are far above the average of all tools (see Figure 4.8). However, when using scenarios this distance is much shorter. This means that, without scenarios, both strengths and limitations of the tools may be masked.

Comparing our results with SAMATE and BSA. As mentioned before, it was not possible to calculate the [DR](#) metric from [SAMATE](#). Therefore, we do not provide ranking of the

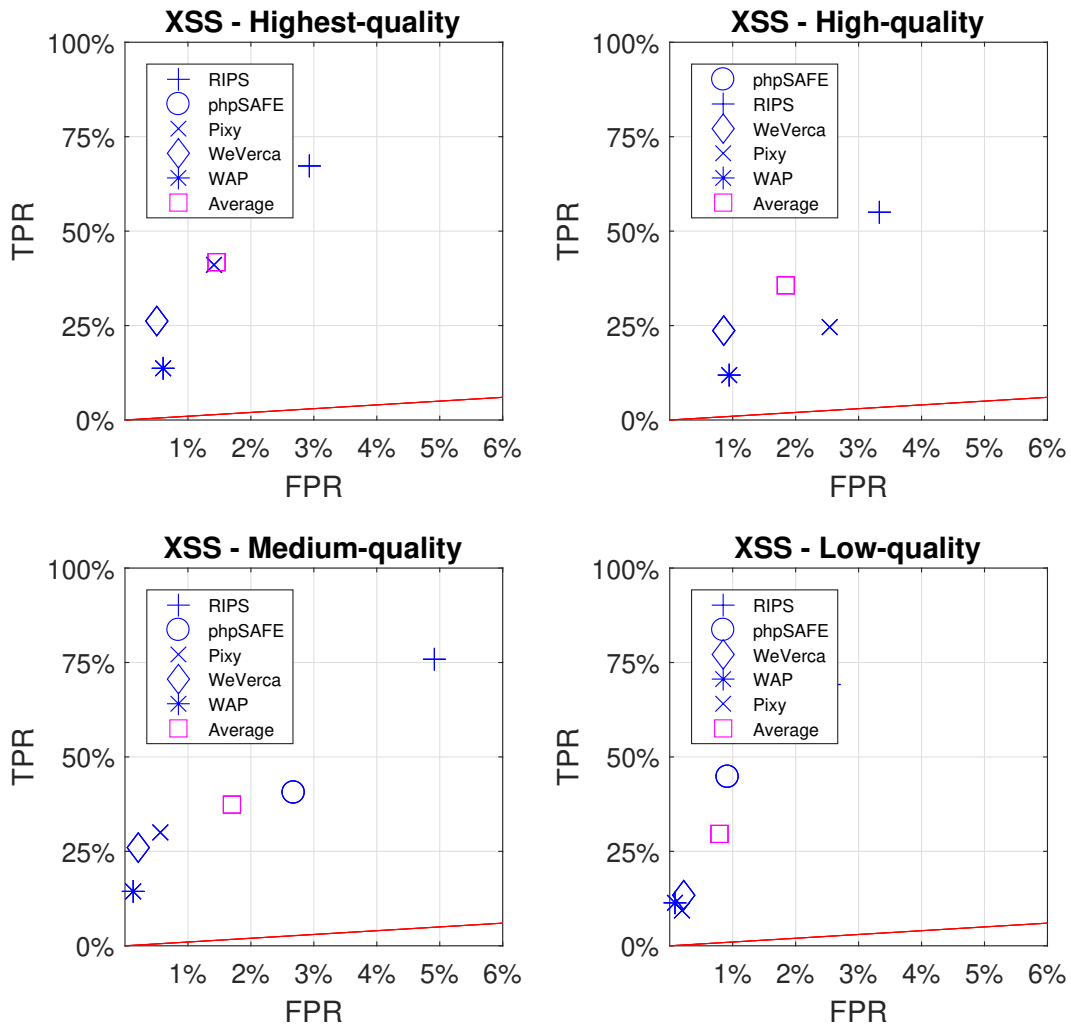


Figure 4.7.: Benchmark XSS comparison by scenario.

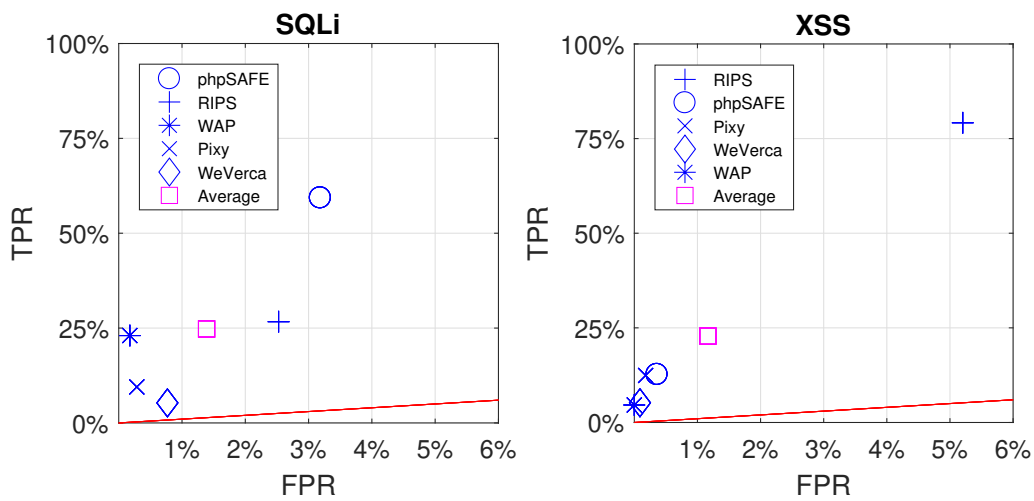


Figure 4.8.: Benchmark SQLi and XSS comparison without scenarios.

Table 4.13.: Ranking of Tools Considering all Plugins and all our Main Metrics.

SQLi							
<i>Tools</i>	<i>Recall</i>	<i>Tools</i>	<i>Informedness</i>	<i>Tools</i>	<i>F-Measure</i>	<i>Tools</i>	<i>Markedness</i>
phpSAFE	0.59	phpSAFE	0.56	phpSAFE	0.66	WAP	0.84
RIPS	0.27	RIPS	0.24	RIPS	0.38	Pixy	0.72
WAP	0.23	WAP	0.23	WAP	0.37	phpSAFE	0.69
Pixy	0.09	Pixy	0.09	Pixy	0.17	RIPS	0.52
WeVerca	0.05	WeVerca	0.05	WeVerca	0.10	WeVerca	0.39
XSS							
RIPS	0.73	RIPS	0.63	RIPS	0.73	WeVerca	0.78
phpSAFE	0.63	phpSAFE	0.48	phpSAFE	0.63	phpSAFE	0.75
Pixy	0.40	Pixy	0.25	Pixy	0.40	RIPS	0.73
WeVerca	0.38	WeVerca	0.23	WeVerca	0.38	WAP	0.72
WAP	0.23	WAP	0.13	WAP	0.23	Pixy	0.70

Table 4.14.: Ranking of Tools Considering all Plugins and **BAS** Metrics.

SQLi				XSS			
<i>Tools</i>	<i>TPR</i>	<i>FPR</i>	<i>BAS</i>	<i>Tools</i>	<i>TPR</i>	<i>FPR</i>	<i>BAS</i>
phpSAFE	59.3	3.2	56.2	RIPS	67.1	3.9	63.1
RIPS	26.7	2.5	24.2	phpSAFE	50.2	2.0	48.2
WAP	23.0	0.2	22.9	Pixy	26.1	1.1	25
Pixy	9.5	0.3	9.2	WeVerca	23.7	0.4	23.3
WeVerca	5.3	0.8	4.5	WAP	13.1	0.4	12.7

tools using this metric. Next, we compare our results with the remaining **SAMATE** metrics (*Precision*, *F-Measure*, and *Recall*).

The **SAMATE** *Recall* and *F-Measure* metrics rank first the same **SAST** tools for all scenarios and classes of vulnerabilities as our main metrics, with the exception of the low-quality scenario in which recall ranks a different **SAST** tool for both classes of vulnerabilities and *F-Measure* ranks a different **SAST** tool for **XSS** vulnerabilities. Comparing with our benchmark, the **SAMATE** *Precision* metric, for **SQLi**, ranks the same **SAST** tools for the highest-quality and low-quality scenarios and different **SAST** tools for the other scenarios. For **XSS**, it ranks the same **SAST** tool for the high-quality scenario and a different **SAST** tool for the other scenarios. The **SAMATE** *Precision* metric reveals that **WAP** is more precise for **SQLi**, and **WeVerca** for **XSS**. However, unlike our main metric, *Markedness*, the *Precision* is useless for ranking the tools as it does not consider the vulnerabilities that were left undetected by the tools. In fact, *Precision* is based on the results (**TPs** and **FPs**) of the **SAST** tools instead of the overall vulnerabilities in the workload. For instance, a tool reporting only one vulnerability and zero **FPs** has 100% *Precision*.

Regarding **BSA**, the results of ranking the tools using the **BAS** metric are similar to the ranking using our metrics, except in the low-quality scenario for both **SQLi** and **XSS**, where the ranking of the tools is different. In short, the **BAS** metric does not provide useful information to the users when they need to choose tools with different security requirements. In fact, tools with the same **BAS** might have different **TPRs** and **FPRs**. However, in projects with more demanding security requirements, the priority may be to find as many vulnerabilities as possible. On the other hand, for projects with tight budgets and where the security is not important, the priority

may be to limit the number of results to observe.

Using as workload all the plugins without distributing them across the scenarios, leads our metrics and the **BAS** metric to rank first the same tools (**phpSAFE** for **SQLi** and **RIPS** for **XSS**), except for the markedness metric where we obtained the **WAP** tool for **SQLi** and the **WeVerca** tool for **XSS**. Thus, we can conclude that organizing the workload in scenarios and defining metrics according to their goals is useful since it allows exploring the capabilities of the tools in different contexts. For example, code with poor quality may have unfeasible execution paths, which requires more sophisticated analysis to avoid **FPS**. For instance, a first run of commercial **RIPS** on **OWASP** Benchmark v1.2 reported 30% of **FPS**. Almost half of them were due to vulnerabilities triggered by code that is semantically unreachable. **RIPS** was improved by disregarding taints that flow through unreachable code and a run with a new version of **RIPS** (30 March 2020) achieved 100% **TP** and 0% **FP**.

The results of running our benchmark show that the proposed approach can be used to successfully rank **SAST** tools in different scenarios. In the next two sections, we discuss the key properties of the benchmark instantiation for WordPress plugins and **SQLi** and **XSS** vulnerabilities.

4.3.3. Limitations and Benchmark Properties

To validate our benchmarking approach, we need to discuss its four main components. The *scenarios* and *metrics* were previously addressed by Antunes et al. [159]. The *procedure* is well-known and follows existing approaches on performance and dependability benchmarking. The *workload* is the component that influences most the results, so it should be discussed in greater detail. The proposed process to build the workload allows selecting real applications with known vulnerabilities. The instantiation of the benchmark approach and the results of the experiments show that it is feasible, but has some limitations/difficulties. The following discuss the main issues regarding the workload definition:

- 1) **Identifying and Collecting Vulnerable Applications.** Since there are many plugins with vulnerabilities, the likelihood of finding plugins with documented vulnerabilities is very high. In fact, results showed that our approach allows the identification of many vulnerable plugins with available source code. However, we also observed that in the **WPVD** there are many vulnerabilities with incomplete documentation, which is needed to evaluate **SAST** tools, such as the vulnerable file, the **LOC**, the vulnerable variable, and **PoC**. In fact, due to this lack of data, the initial number of plugins identified was dramatically reduced from 273 to 134. This problem can be minimized by using more vulnerability databases.
- 2) **Assigning Applications to Scenarios.** We observed that our workload is unbalanced concerning the number of plugins by scenario (see Section 4.2.2). However, this was expected as the number of plugins collected (134) is not very high and the real percentages of plugin with five or one stars is very low. Moreover, the distribution of the plugins by scenario seems to follow a pattern similar to a normal distribution (the extreme scenarios with less plugins and the middle scenarios with much more plugins). Adding more plugins to the workload could help mitigating this issue.
- 3) **Identifying **VLOCs** and **NVLOCs**.** The process used to identify the lists of **VLOCs**

and **NVLOCs** requires updating values during the process of benchmarking when the tools report previously unknown vulnerabilities. This occurred for 2 out of 3 tools, and the total number of manual reviews required was 251 (WAP 168; WeVerca 83). Therefore, as the number of tools benchmarked increases, the number of required reviews may decrease due to the overlap of vulnerability detection between the tools. Moreover, none of the tools reported vulnerabilities in **LOCs** outside the lists of **NVLOCs** and **VLOCs**. This means that the process of identifying the **NVLOCs** can be trusted.

The **SAST** tools failed analyzing several of files of the plugins. We observed that the percentage of **LLOC** where at least one **SAST** tool failed the analysis decreases as the quality of the code increases. The percentages are 56% for the low-quality scenario, 50% for the medium-quality scenario 35%, for the highest-quality scenario, and 38% for the high-quality scenario. This shows that the code with better quality (i.e., less complex, as recommended by the participants in the SwMM-RSV NIST’s workshop [180]) increases the probability of being successfully analyzed by the **SAST** tools. As a side effect, this contributes to reducing vulnerabilities in the software, since they are more likely to be detected by the **SAST** tools. Because users have different requirement constraints regarding the code quality and this in turn has a direct impact on the ability to detect vulnerabilities, it is very important to have the benchmark configured for scenarios based on the internal software quality.

It is important to emphasize that, to be accepted, any benchmark should fulfill a set of key properties: representativeness, portability, repeatability, scalability, non-intrusiveness, and simplicity of use [151] [156]. In the following, we discuss these properties:

- 1) **Representativeness.** Our workload includes real applications since it is composed by WordPress plugins widely used in different scenarios, with real vulnerabilities. However, the workload across the various scenarios is unbalanced, which may affect the results in some cases. For example, in the highest-quality scenario and for **SQLi**, only two **SAST** tools reported vulnerabilities, which may limit our study. Works using other tools are needed for improving the characterization of the vulnerable/non-vulnerable **LOCs** in the workload. Trust on the representativeness of the metrics is increased by previous works that showed that the different metrics should be considered for different vulnerability detection scenarios [158] [106].
- 2) **Portability.** **SAST** tools do not need to run the program being analyzed, so the benchmark can be used for evaluating different **SAST** tools able to detect **SQLi** and **XSS** vulnerabilities in PHP code, as demonstrated in the experiments. Addressing other languages and classes of vulnerabilities requires defining a new workload, following the process proposed.
- 3) **Repeatability.** **SAST** tools with the same settings always produce the same results as they analyze the static program structure in a deterministic way, making the results of the tools deterministic. We also verified this property empirically by running several times the tools and obtaining the same results.
- 4) **Scalability.** The workload can be scaled in the number and in the complexity of the tests, since the load increases proportionally, not exponentially. The benchmark can be applied without any change to the **SAST** tools with different functionalities and maturity. In fact, it is necessary to update the list of **P** and **N** instances when these new tools report

vulnerabilities in unchecked **SSs** in the list of **N** instances. Furthermore, as **SAST** tools may fail in analyzing large files can occur, issue may arise in the characterization of the workload in terms of **VLOCs**. For instance, in the characterization of the workload, it was necessary to ensure that all files were analyzed by at least one of the tools used in the process.

- 5) **Non-intrusiveness.** Our approach is non-intrusive, as it does not require any change to the **SAST** tools under benchmarking.
- 6) **Simplicity of Use.** Running the benchmark takes simple steps: i) configuring and executing the **SAST** tools for searching vulnerabilities in the workload; ii) normalizing the result of the **SAST** tools to a common format (depending on the output format options of the **SAST** tool, more or less effort is required to convert their results to a common format, but this task can be somehow automated by developing one wrapper for each **SAST** tool); iii) comparing the results of the **SAST** tools with known vulnerable and non-vulnerable **LOCs**; and iv) calculating the metrics and ranking the tools. These are quite straightforward, although time consuming in some cases, due to some amount of manual work involved (to verify new vulnerabilities).

The output format of the **SAST** tool should be harmonized in order to automate the analysis and to be able merge the results of several **SAST** tools. For example, for one plugin we found over 15 vulnerabilities where a **SS** occupied several **LOCs** outputting individual array elements or arrays contained in fields of objects. For these cases, we found that the **SAST** tools report the vulnerabilities in different **LOCs** which makes the comparison of **SAST** tools somewhat cumbersome.

A key aspect is that, although almost all stages of the benchmarking process can be automated, the *Vulnerability Verification* stage (Section C.4) is a manual process. It is a difficult and time consuming task and depends on the availability of security experts. In fact, there is a need for tools to improve the vulnerability verification by reducing the reliance on labour intensive and potentially error prone analysis by experts. Automating the vulnerability verification stage is a challenging task with a key question: *does the vulnerability exists and is exploitable?* This is addressed in Chapter 6.

4.4. Conclusion

In this chapter, we addressed the problem of choosing adequate **SAST** tools for vulnerability detection in web applications. First, we proposed an approach to design benchmarks for evaluating such **SAST** tools considering different levels of criticality. It includes the definition of the main components of a benchmark for **SAST** tools: scenarios, metrics, workload, procedure, and experimental setup.

To evaluate our proposed approach, we created a benchmark for WordPress plugins and tested it with five free **SAST** tools searching for **XSS** and **SQLi** vulnerabilities in 134 WordPress plugins with real vulnerabilities, developed in **PHP**. The experimental results showed that the best tool changes from one scenario to another and also depends on the class of vulnerabilities being detected. Our novel benchmark approach is a valuable tool to help project managers choosing

the best [SAST](#) tool according to their needs and the resources available.

The comparison of the results using our metrics and the metrics from [SAMATE](#) and [BSA](#) reveals that the use of the same metrics for all scenarios makes more difficult the choice of the most appropriated tool for a project with specific requirements of security. For instance, the Discrimination Rate ([DR](#)) and Benchmark Accuracy Score ([BAS](#)) metrics may mask the capabilities of the tools when a tool reports [FPs](#). Therefore, the metrics should be chosen according to the vulnerability detection scenario. Moreover, we found that identifying the [TPs](#) in the workload is fundamental to better characterize the tools. However, since the number of [N](#) in real applications might be much higher than the number of [P](#), the metrics should be improved to balance the weight of the [TPs](#) and [FPs](#) in the computation of the metrics.

Static Application Security Testing ([SAST](#)) tools still need a lot of improvements to become better in catching implementation security bugs. Since new methods are continually emerging, benchmarks can quickly become out of date. To avoid this, a benchmark should be extensible [[154](#)], but this may require a significant additional effort. In fact, “continuous benchmarks”, which are continually updated, are especially convenient. For example, the [OWASP](#) Benchmark has been improved with the contributions from researchers, users and [SAST](#) tools developers. The contributions include new benchmark test cases and [SAST](#) tools results using the [OWASP](#) Benchmark suite.

Combining Diverse SAST Tools for Web Security

State-of-the-art **SAST** tools are, on average, able to detect about half of the existing security vulnerabilities [1]. To improve their overall detection capabilities, some researchers have proposed combining the results of multiple **SAST** tools [110][112][111]. Of particular interest is the work of Diaz et al. [116], which compares the performance of nine **SAST** tools, most of them commercial tools, against the NIST SAMATE (Software Assurance Metrics And Tool Evaluation) Reference Dataset test suite [117]. Based on the results, the authors recommended the use of several tools with different designs and detection algorithms and/or heuristics to improve detection. On the other hand, Beller et al. [118] investigated how common is the use of **SAST** tools in real-world, taking as reference the 122 most popular Open-Source Software projects. They observed that a single **SAST** tool was used in 41% of the projects, two **SAST** tools in 22% of the projects, and three **SAST** tools in only 14% of the projects. This suggests that developers might not be aware of the benefits of using multiple tools and/or that the increase of false positives reported may lead developers to avoid using multiple **SAST** tools [119]. However, existing works are limited in several aspects: the workloads are synthetic or just a small set of applications, the evaluation metrics used are too simple (e.g., number of TPs), and the analysis does not consider the specificity of the development scenarios where the tools are to be used, which may vary both in terms of development time and resources.

In this chapter we present two case studies on combining the results of several **SAST** tools. The case studies are based on the results of benchmarking five free **SAST** tools to detect **SQLi** and **XSS** vulnerabilities in a workload composed by 134 WordPress plugins, organized in four vulnerability detection scenarios (see Chapter 4.2 for more details). In the first case study, we combine the results of the **SAST** tools using a *1-out-of- n* ¹ adjudication. As one limitation of *1-out-of- n* adjudication is the potential increase of **FPs**, which may be unacceptable in many

¹Raise an alarm for a vulnerability when any of n **SAST** tools in the diverse configuration does so.

situations, in the second case study, we study at all the possible *1-out-of-n*, *n-out-of-n²* and *majority voting*³ adjudications. This way, we get more evidence on the interplay between FPs and FNs in diverse SAST configurations. We also present an in-depth analysis of the code of the WordPress plugins to find reasons for the diversity in the results of the SAST tools. The results also highlight where SAST developers should look in order to improve them.

The reasoning for separating the two case studies is as follows. SAST tools sometimes fail to properly analyze some source code files of a specific project. This occurred during the creation of the dataset used in this chapter (see Section 4.3.1 for more details). The *1-out-of-n* strategy uses the “OR” boolean operator to combine results, which allows broadening the results. Thus, if a tool fails in a file it does not impede the combination, as shown in the first case study. However, when combining the results using *n-out-of-n* and *majority voting* strategies, a failure in a tool when analyzing a file makes the results not comparable, as these strategies use the “AND” boolean operator to combine the results (this operator narrows the results). In practice, if we combine the results of the N tools where one fails, we are eliminating all the combinations that include that tool. This way, to make the analysis comparable, in the second case study, we consider only the results obtained from the files that could be successfully analyzed by all five tools. Furthermore, although the dataset is organized in four scenarios, now we discarded the scenarios, since our goal is now focused on the interplay between FPs and FNs.

We also present a third case study combining the results of the five SAST tools using a *1-out-of-n* adjudication under a dataset composed by synthetic test cases. Because this case study is not in the center of our contributions, we present it in Appendix E-Case Study: Synthetic Dataset Using the 1-out-of-n Strategy, where we compare its results with the ones from Section 4.2.

The organization of the chapter is as follows. Section 5.1 presents the case study combining the results of SAST tools using a *1-out-of-n*. Section 5.2 presents the case study combining the results of SAST tools using *1-out-of-n*, *n-out-of-n* and *majority voting*. Section 5.3 addresses the threats to validity of the case studies in this chapter. Section 5.4 concludes the chapter.

5.1. Case Study: 1-out-of-n Adjudication

In this work, we argue that the use of multiple SAST tools may be helpful, as more vulnerabilities are likely to be reported, however, the drawback is that the number of FPs may, at the same time, increase. Furthermore, we also claim that the acceptable/expected outcome of the static analysis process (in terms of coverage and FPs) depends on the application scenario (see Section 4.1.1). Thus, it is no longer evident that combining more SAST tools is better in every case. To confirm/negate these claims, we studied the potential of combining the outputs of multiple SAST tools with a *1-out-of-n* strategy as a way to improve the performance of vulnerability detection across different realistic development scenarios.

The following sections present the formulation of the hypotheses, the analysis approach, the

²Raise an alarm for a vulnerability only when all n SAST tools in the diverse configuration do so.

³Raise an alarm for a vulnerability when the majority of the n SAST tools in a diverse configuration do so.

results for [SQLi](#) and [XSS](#) vulnerabilities and the responses to the hypotheses.

5.1.1. Hypotheses and Analysis Approach

To drive the case study, we formulate the following four hypotheses:

H_1 : the number of vulnerabilities detected always increases as the number of combined [SAST](#) tools increases.

H_2 : the number of [FPs](#) always increases as the number of combined [SAST](#) tools increases.

H_3 : the best combination of [SAST](#) tools is the same across all development scenarios.

H_4 : the best combination of [SAST](#) tools is the same across different classes of vulnerabilities.

Although the response to the above hypotheses may seem obvious, empirical evidences are missing in the literature to better understand the advantages and limitations of different combinations of [SAST](#) tools, when considering representative vulnerability detection scenarios. For example, a less informed researcher or developer could easily state that the number of vulnerabilities detected increases as the number of combined [SAST](#) tools increases, however, he misses knowledge about which scenarios that applies, and to which amount that happens for different types of vulnerabilities, and what is the impact in terms of [FPs](#). All of these are aspects that require more detailed studies, as the one presented here.

To fully test our hypotheses, all combinations of tools should be considered for each scenario and class of vulnerabilities. As mentioned before, the results of [SAST](#) tools are combined using *1-out-of-n* strategy, which raise an alarm for a vulnerability when any one of n [SAST](#) tools does so. The process proposed to calculate the combined results for two or more [SAST](#) tools is based on a set of steps that we automated (see Figure 5.1):

- 1) **Calculate the number of P and N in the workload:** using the lists of [VLOCs](#) and [NVLOCs](#), and the distribution of the plugins per scenario, calculate the number of P and the number of N for each scenario and for each class of vulnerability, as described in Section 4.1.2.
- 2) **Combine results of [SAST](#) tools:** for each scenario, class of vulnerability and combination of [SAST](#) tools, merge the outputs of the tools discarding the duplicated [TPs](#) and [FPs](#).
- 3) **Calculate the combined confusion matrices:** with the outputs from 1) and 2) calculate, for each scenario, class of vulnerability and combination of [SAST](#) tools, the corresponding confusion matrix (made up of [TP](#), [FP](#), [FN](#), and [TN](#)).
- 4) **Calculate the metrics and rank the combinations of tools:** with the results from 3), compute the metrics recommended for each scenario (see Table 4.1) and rank the combinations of [SAST](#) tools.

With the five individual [SAST](#) tools included in the case study, there are 31 possible combinations of the results using a *1-out-of-n* strategy ($C_1^5 + C_2^5 + C_3^5 + C_4^5 + C_5^5 = 31$). We built 10 combination pairs ($C_2^5 = 10$), 10 combination triplets ($C_3^5 = 10$), 5 combination quadruples ($C_4^5 = 5$), one quintet ($C_5^5 = 1$) [SAST](#) tools combination and the five individual tools. To simplify the presentation of the results we assigned a character to each tool: a - [phpSAFE](#), b - [RIPS](#), c -

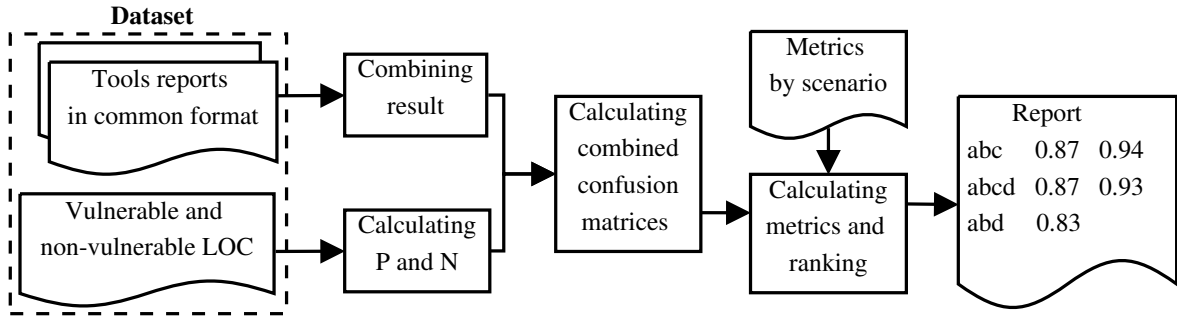


Figure 5.1.: Overall process of combining the results of multiple tools

WAP, d - Pixy, and e - WeVerca.

The results for each combination of the five **SAST** tools, organized by scenario and type of vulnerability, **SQLi** and **XSS**, are presented in Table 5.1. Columns *TP*, *FP*, *FN*, and *TN* show the confusion matrix for the corresponding combination. The table shows only the TOP 5 (of 31) combinations of **SAST** tools for each scenario and type of vulnerability. The results for all combinations can be consulted in Appendix D-Results for all Combinations of five **SAST** Tools: **WordPress Plugins** and are also online available at [197]. Table 5.1 also includes the ranking of the individual tools, as reference. The data are firstly ordered by the main metric (column Metric), secondly by the tiebreak metric (column Tiebreaker), and finally by the number of **SAST** tools in the combination. For the same results, a solution with less tools is preferable because running less tools requires less effort.

5.1.2. Results for **SQLi** Vulnerabilities

The goal of the highest-quality scenario is to find the highest number of vulnerabilities, even if reporting many **FPs**. Since there are several solutions, both with the same number of vulnerabilities and the same number of **FPs**, the best solution is the solution *ac*, as it has the least number of **SAST** tools (Table 5.1). We see that **RIPS**, **Pixy** and **WeVerca** did not found vulnerabilities and did not report **FPs**. In this scenario, there are 2 **POP** plugins and 10 **OOP** plugins with 43% of **POP** code (see Table 4.7). Thus, the poor results of **RIPS** and **Pixy** are probably because they are not prepared to analyze **OOP** code.

In the high-quality scenario, the combination that ranks first is *acde*, with the highest number of vulnerabilities found (318). In 2nd place there is a similar solution (*abce*) with the same number of vulnerabilities, but with one more **FP**. Individually, we see that **phpSAFE** found many vulnerabilities but also many **FPs**, while the other tools report much less **FPs** along with much less vulnerabilities too.

In the medium-quality scenario, the combination of **SAST** tools that ranks first is *abce* with both the highest number of vulnerabilities found (251) and **FPs** (163). The solutions in the 2nd to 8th positions, have the same number of **FPs** and successively less vulnerabilities detected. In 9th place there is a solution composed by three **SAST** tools (*acd*) with about 2/3 of the vulnerabilities found and 2/5 of **FPs**. In terms of vulnerabilities detected, this solution performs similarly to **RIPS** (*b*) individually. However, **RIPS** reports about the double of **FPs**. This means that, the best

Table 5.1.: Best Solutions for the WordPress plugins: SQLi, XSS and SQLi + XSS

SQLi							XSS							SQLi + XSS				
Tools	TP	FP	Pg	MM	TM	P/R	Tools	TP	FP	Pg	MM	TM	P/R	Tools	MM	TM		
<i>Highest-quality</i>			<i>Recall</i>	<i>Prec.</i>			<i>Recall</i>			<i>Prec.</i>			<i>Rec.</i>			<i>Prec.</i>		
ac	65	5	9	0.867	0.929	-	ab	165	43	11	0.982	0.793	-	abc	0.947	0.821		
ace	65	5	9	0.867	0.929	-	abe	165	43	11	0.982	0.793	-	abcd	0.947	0.821		
abce	65	5	9	0.867	0.929	-	abc	165	45	11	0.982	0.786	-	abcde	0.947	0.821		
acde	65	5	9	0.867	0.929	-	abd	165	45	11	0.982	0.786	-	abce	0.947	0.821		
abc	65	5	9	0.867	0.929	-	abce	165	45	11	0.982	0.786	-	acde	0.844	0.869		
c	49	4	7	0.653	0.925	-	b	113	29	10	0.673	0.796	-	a	0.539	0.851		
a	29	5	5	0.387	0.853	-	a	102	18	8	0.607	0.850	-	b	0.465	0.796		
e	0	0	0	0	-	-	d	69	14	7	0.411	0.831	-	c	0.296	0.878		
b	0	0	0	0	-	-	e	44	5	7	0.262	0.898	-	d	0.284	0.831		
d	0	0	0	0	-	-	c	23	6	3	0.137	0.793	-	e	0.181	0.898		
<i>High-quality</i>	<i>Informedness</i>			<i>Rec.</i>	<i>Prec.</i>			<i>Informedness</i>			<i>Rec.</i>	<i>Prec.</i>			<i>Informedness</i>	<i>Rec.</i>		
acde	318	59	36	0.866	0.919	0.844	abce	1841	224	51	0.961	10.0	0.892	abce	0.946	0.987		
abce	318	60	36	0.865	0.919	0.841	abcde	1841	224	51	0.961	10.0	0.892	abcde	0.946	0.987		
abcde	318	60	36	0.865	0.919	0.841	abe	1838	223	51	0.960	0.998	0.892	abde	0.930	0.971		
ace	316	59	36	0.860	0.913	0.843	abde	1838	223	51	0.960	0.998	0.892	abe	0.930	0.971		
acd	311	58	35	0.847	0.899	0.843	abc	1770	224	51	0.923	0.961	0.888	abc	0.910	0.951		
a	274	58	30	0.740	0.792	0.825	a	1164	90	46	0.617	0.632	0.928	a	0.636	0.657		
c	44	4	12	0.124	0.127	0.917	b	1013	194	46	0.517	0.550	0.839	b	0.454	0.483		
b	43	2	8	0.123	0.124	0.956	e	436	50	25	0.228	0.237	0.897	e	0.200	0.207		
e	18	1	6	0.051	0.052	0.947	d	453	148	28	0.221	0.246	0.754	d	0.193	0.214		
d	16	0	7	0.046	0.046	10.0	c	219	55	18	0.110	0.119	0.799	c	0.112	0.120		
<i>Medium-quality</i>	<i>F-Measure</i>			<i>Rec.</i>	<i>Prec.</i>			<i>F-Measure</i>			<i>Rec.</i>	<i>Prec.</i>			<i>F-Measure</i>	<i>Rec.</i>		
abce	251	163	21	0.737	0.940	0.606	abce	2386	652	46	0.879	0.999	0.785	abce	0.863	0.993		
abcde	251	163	21	0.737	0.940	0.606	abcde	2386	652	46	0.879	0.999	0.785	abcde	0.863	0.993		
abc	250	163	21	0.735	0.936	0.605	abc	2383	652	46	0.879	0.998	0.785	abc	0.863	0.991		
abcd	250	163	21	0.735	0.936	0.605	abcd	2383	652	46	0.879	0.998	0.785	abcd	0.863	0.991		
abde	237	163	19	0.711	0.888	0.593	abde	2359	652	46	0.874	0.987	0.783	abde	0.856	0.977		
b	153	113	6	0.574	0.573	0.575	b	1812	490	43	0.773	0.759	0.787	b	0.752	0.740		
a	99	50	15	0.476	0.371	0.664	a	970	267	41	0.535	0.406	0.784	a	0.529	0.402		
c	72	0	11	0.425	0.27	10.0	d	717	56	23	0.454	0.300	0.928	d	0.441	0.290		
d	54	13	4	0.323	0.202	0.806	e	621	21	19	0.410	0.260	0.967	e	0.383	0.242		
e	21	34	3	0.130	0.079	0.382	c	344	13	18	0.251	0.144	0.964	c	0.270	0.157		
<i>Low-quality</i>	<i>Markedness</i>			<i>Prec.</i>	<i>Rec.</i>			<i>Markedness</i>			<i>Prec.</i>	<i>Rec.</i>			<i>Markedness</i>	<i>Prec.</i>		
bc	6	0	2	0.963	10.0	0.120	c	62	3	6	0.835	0.954	0.114	c	0.857	0.957		
bcd	6	0	2	0.963	10.0	0.120	abc	542	117	12	0.822	0.823	0.994	cde	0.835	0.925		
bce	6	0	2	0.963	10.0	0.120	abcd	542	117	12	0.822	0.823	994	ce	0.832	0.925		
bcde	6	0	2	0.963	10.0	0.120	abce	543	117	12	0.822	0.823	0.996	cd	0.819	0.916		
c	5	0	2	0.962	10.0	0.100	abcde	543	117	12	0.822	0.823	0.996	de	0.815	0.911		
c	5	0	2	0.962	10.0	0.100	c	62	3	6	0.835	0.954	0.114	c	0.857	0.957		
b	1	0	1	0.959	10.0	0.020	a	244	33	10	0.803	0.881	0.448	e	0.802	0.901		
a	36	32	7	0.517	0.529	0.720	e	73	8	7	0.785	0.901	0.134	d	0.776	0.879		
e	0	0	0	-	-	0.000	b	377	91	10	0.760	0.806	0.692	b	0.761	0.806		
d	0	0	0	-	-	0.000	d	51	7	9	0.758	0.879	0.094	a	0.748	0.812		
abce*	675	260	-	-	0.722 ¹	0.915 ²	abce*	4935	1038	-	-	0.826 ¹	0.998 ²	abcde*	0.812 ¹	0.987 ²		

Tools: a - phpSAFE, b - RIPS, c - WAP, d - Pixy, e - WeVerca. *Solution with best recall regardless the scenarios.

Pg - Number of plugins where the combination of **SAST** tools reports vulnerabilities.

MM - Main Metric, TM - Tiebreaker Metric, Rec. - Recall(R)⁽²⁾, Prec. (P)⁽¹⁾ - Precision.

individual **SAST** tool (*a*) can be replaced by a combination of **SAST** tools (*acd*) that individually perform worse than it does, but together perform better.

The best solution for the low-quality scenario is *bc*, with 6 **TPs** and zero **FPs**. Since the resources available for fixing vulnerabilities in this scenario are very limited, this solution fits very well its goal. Note that, **phpSAFE** individually (**SAST** tool *a*) reports six times more vulnerabilities (36), but also many more **FPs** (32), which is not desirable in this case.

5.1.3. Results for XSS Vulnerabilities

For the highest-quality scenario, and focusing on **XSS** vulnerabilities (see Table 5.1), the best solution is the combination *ab*, which detected the highest number of vulnerabilities (165). In the 2nd position, there is a combination of three **SAST** tools (*abe*) that detected the same number of vulnerabilities and **FPs** (43). A key observation is that the combination *ab* detected the same vulnerabilities as **WeVerca** (*e*) individually, as well as other vulnerabilities. The solutions from the 3rd place to the 5th place detected the same number of vulnerabilities of the two first solutions, but reported two more **FPs**. The number of **SAST** tools in these solutions varies from two to five. This clearly shows that adding a tool to an existing solution does not always increase the number of vulnerabilities found.

In the high-quality scenario, the best combination of **SAST** tools is *abce*, which has the highest number of vulnerabilities detected (1841), but also has the highest number of **FPs** (224). In this case, every **FP** decreases the informedness metric by 0.02% (1/5825), while every **TP** increases the metric by 0.05% (1/1842). Therefore, all the **FPs** reported decrease the metric only 3.56% (227/5825). For that reason, the best solution is the one with both the highest number of **TPs**, although it may have a large number of **FPs**.

The best combination of **SAST** tools to detect **XSS** vulnerabilities in the medium-quality scenario is also *abc*. We see that this combination of **SAST** tools is better than the others because it has the highest number of **TPs**, although it also has the highest number of **FPs**. This is because the recommend metric, *F-Measure*, does not consider the **TNs** and it considers the **TPs** more important than the **FPs**.

SAST tool *c* (**WAP**) comes alone in the 1st position for the low-quality scenario. It detected 62 **TPs** and reported only 3 **FPs**. This means it has both a high precision and an inverse precision. This way, the resources will be consumed in fixing vulnerabilities, as desired for this scenario, instead of being wasted confirming many **FPs**. However, from the 2nd to 8th positions there are combinations of **SAST** tools that detected about ten times more **TPs** but also more than thirty times more **FPs** than **SAST** tool *c*. For example, the solution in the 2nd position, *abc*, detected 543 **TPs** but also reported over 39 times more of **FPs** (117), which is not acceptable for this scenario. Like for **SQLi** vulnerabilities, in this same scenario, the **SAST** tool *a* (**phpSAFE**) reported both many vulnerabilities and many **FPs**. Finally, unlike for **SQLi** vulnerabilities, tool *b* (**RIPS**) reported both the highest number of **TPs** and **FPs**. This shows that the same tool may have a different performance depending on the type of vulnerability being detected.

It is important to emphasize that, considering the top five combinations presented for each

scenario, the **SAST** tool *a* (phpSAFE) is included in all of them, except for the low-quality scenario, **SAST** tool *b* (RIPS) is included in five solutions, **SAST** tool *c* (WAP) in seven, **SAST** tool *d* (Pixy) in one, and **SAST** tool *e* (WeVerca) in four. **SAST** tool *c* is the tool that reported less **FPs** in almost all scenarios, despite it is not the one that found more vulnerabilities. Thus, the effectiveness of existing solutions has a high probability to improve when **SAST** tool *c* is added to these solutions. **SAST** tools *a* and *b* report both many vulnerabilities and **FPs**. The individual ranking of the **SAST** tool *d* is always below the middle of the ranking and it is the worst of all in 4 of 8 cases. This is probably because tool *d* is already an old tool and it is, therefore, not prepared for analyzing **OOP** code. However, despite being recent and prepared for **OOP** code, **SAST** tool *e* has a performance similar to *d*, so it has room for improvements.

5.1.4. Testing the Hypotheses

Based on our findings, all hypotheses stated in the Section 5.1.1 are false. Hypothesis H_1 (the number of vulnerabilities detected always increases as the number of combined **SAST** tools increases) is false because we found many cases where adding a **SAST** tool to an existing combination does not increase the number of vulnerabilities found (e.g., for the highest-quality scenario and **XSS**: *ab*, *abe*, *abce*).

We observed that the number of **FPs** does not always increase with the number of **SAST** tools in a combination (e.g., for the medium-quality scenario and **SQLi**: *abc*, *abcd*, *abcde*). Therefore, hypothesis H_2 (the number of false positives always increases as the number of combined **SAST** tools increases) is also false. As there is frequently an overlap between the **FPs** reported by different **SAST** tools, in some cases combinations with more tools can detect more vulnerabilities, while maintaining the same number of **FPs**. Also note that, none of the best combinations includes all **SAST** tools.

The best solution for vulnerability detection depends on the chosen scenario and on class of vulnerability. Therefore, hypotheses H_3 (the best combination of **SAST** tools is the same across development scenarios) and H_4 (the best combination of **SAST** tools is the same across different classes of vulnerabilities) are both false. In fact, the detection capabilities of the **SAST** tools are not uniform across the two classes of vulnerabilities. The same occurs for combinations of **SAST** tools. Moreover, in almost all cases the values of the metrics for **XSS** vulnerabilities are better than for the **SQLi** vulnerabilities.

In summary, the main advantage of combining the results of several **SAST** tools is the identification of more vulnerabilities. In fact, for several cases there are tools that individually did not find any vulnerabilities or found few vulnerabilities in many plugins. Moreover, we observed that, even using all the **SAST** tools, some vulnerabilities remain undetected. A key remark is that combining many tools can be counterproductive in some cases as that will not lead to the detection of more vulnerabilities, but will increase the number of **FPs** reported, which then need to be verified manually by the developers. Finally, identifying the strengths and limitations of **SAST** tools, helps developers to determinate how such tools can be combined to provide a more thorough analysis of the software depending on the specificities of the scenario and on the class of vulnerability being analyzed.

5.2. Case Study: Diverse Adjudication Strategies

When using diverse [SAST](#) tool systems, the decision on whether to flag code as vulnerable or not depends on the adjudication of outputs from the individual [SAST](#) tools. In this second case study, we look at three common configurations for adjudication:

- 1) **1-out-of-n** (abbreviated 1ooN): the code is labeled as vulnerable as long as any one of n [SAST](#) tools have labeled it as vulnerable.
- 2) **n-out-of-n** (abbreviated NooN): the code is labeled as vulnerable only if all n [SAST](#) tools in a given configuration of n [SAST](#) tools label the code as vulnerable.
- 3) **Majority voting**: the code is labeled as vulnerable as long as the majority of [SAST](#) tools in a given configuration of n [SAST](#) tools (e.g., 2 out of 3, 3 out of 4, 3 out of 5, etc.) have labeled it as vulnerable.

Different [SAST](#)s, with diverse designs, have a high potential to report more vulnerabilities, although with some overlap when analyzing the same code. Therefore, for each configuration, the results obtained vary in several key aspects: the number of [TP](#)s, the number of [FP](#)s and the confidence in the combined results of the [SAST](#)s. An important challenge that security experts face is trying to identify a remediation strategy that best balances these three competing forces. Note that, although the number of vulnerabilities and number of [FP](#)s in *n-out-of-n* configurations is certainly smaller than in *1-out-of-n* configurations, however, the confidence of the results should be higher. Depending on the resources available for vulnerability fixing, it is extremely important for developers to consider the configuration that best fits their purposes. For instance, in a project with just a few resources for vulnerability remediation, developers may consider primarily a configuration that reports less results but with a higher confidence.

In the previous case study, we found that some of the [SAST](#) tools exhibit considerable diversity in their ability to detect the types of vulnerabilities analyzed. This way, besides analyzing the performance of the different [SAST](#)s configurations as a black box, we study the diversity in detecting vulnerabilities. For this, we performed an extensive analysis of the code of the plugins to generate simple test cases (i.e., small scripts with few PHP constructs (e.g., `if`, `print`)). Next, we ran the five [SAST](#)s tools to detect vulnerabilities in these test cases, and analyzed the results of the five [SAST](#)s tools in more detail to investigate the reasons for the observed diversity in the results. The results show that the effectiveness of the [SAST](#)s tools depends on the types of code constructs that are used. Knowing the reasons of this diversity is an important matter for end-users and developers of [SAST](#) tools.

In summary, the contributions of this case study are as follows: i) we analyze sensitivity and specificity measures for all possible two-, three-, four- and five-[SAST](#) tools diverse configurations; ii) we present empirically supported guidance on which combination of [SAST](#) tools provides most benefits in detecting vulnerabilities, with a reduced [FP](#) rate; iii) we provide a methodology for analyzing in-depth the target code, including a process for generating test cases (i.e., small pieces of code) based on this target code; iv) we present a detailed empirical evaluation of that methodology; and v) we provide results of an in-depth analysis of the target code to better understand the diversity in the design and configuration of these tools that helps to explain their

exhibited diversity in detection.

The following sections present the formulation of the hypotheses and analysis methodology, the analysis of the diversity of the results of the individual **SAST** tools, the main results of the diversity analysis, an in-depth analysis of the plugins and explanations about the possible causes of the observed diversity in the behavior of the **SAST** tools and the response to the hypotheses.

5.2.1. Hypotheses and Analysis Approach

The goal of this experiment is to understand the potential of combining the outputs of multiple **SAST** tools with a *1-out-of-n*, *n-out-of-n* and majority voting strategies and is focused on the studying the interplay between False Positive (**FP**) and False Negative (**FN**) of the combinations. In practice, we formulate the following hypotheses:

- H_1 : the sensitivity obtained for combinations of **SAST** tools using *1-out-of-n* adjudication is higher than the sensitivity obtained using *majority* adjudication.
- H_2 : the sensitivity obtained for combinations of **SAST** tools using *1-out-of-n* adjudication is higher than the sensitivity obtained using *n-out-of-n* adjudication.
- H_3 : the sensitivity obtained for combinations of **SAST** tools using *majority* adjudication is higher than the sensitivity obtained using *n-out-of-n* adjudication.
- H_4 : the specificity obtained for combinations of **SAST** tools using *1-out-of-n* adjudication is lower than the specificity obtained using *majority* adjudication.
- H_5 : the specificity obtained for combinations of **SAST** tools using *1-out-of-n* adjudication is lower than the specificity obtained using *n-out-of-n* adjudication.
- H_6 : the specificity obtained for combinations of **SAST** tools using *majority* adjudication is lower than the specificity obtained using *n-out-of-n* adjudication.

As mentioned before, **SAST** tools sometimes fail to properly analyze the source code files of a specific project. This may be because of memory constraints due to the complexity size and complexity of some files. The tool usually fails by crashing or not providing a result. Overall, as shown in Section 4.3.1, phpSAFE was unable to analyze 130 files, RIPS could not analyze 1,473 files for **SQLi** and 19 for **XSS**, Pixy did not process 1,473 files, and WeVerca was not able to analyze a total of 20 files. To make the analysis comparable, we consider only the results obtained from the files that could be successfully analyzed by all five tools. This reduces the number of files we can include in the study, but enables us to have a common subset of files that were analyzed by all the tools, making the results of the tools comparable. Furthermore, although the dataset is organized in four scenarios, now we discarded the scenarios, since our goal is focused on the study of the interplay between **FPs** and **FNs**.

Considering only the files successfully analyzed by all five tools, the dataset contains a total of 392,377 **LLOC** (i.e., commented and whitespace lines are excluded), as can be seen in Table 5.2. Since programming orientation may be relevant for the performance of the **SAST** tools, the table shows the **LLOCs** that have only **POP** code and those that have **OOP** code. The table also shows the **VLOCs** and **NVLOCs** for **SQLi** and **XSS**.

Table 5.2.: Dataset information by class of vulnerability and type of code.

Class	Code Type	Plugins	Files	LLOC			VLOC	NVLOC	Total
				POP	OOP	Total			
SQLi	POP	30	259	21,501	0	21,501	206	651	857
	OOP	87	1,909	89,575	46,617	136,192	438	4,178	4,616
	Total	117	2,168	111,076	46,617	157,693	644	4,829	5,473
XSS	POP	30	269	21,643	0	21,643	3,298	12,479	15,777
	OOP	100	3,132	154,104	58,937	213,041	1,051	3,574	4,625
	Total	130	3,401	175,747	58,937	234,684	4,349	16,053	20,402

In practice, we analyze the results of the **SAST** tools using all possible combinations based on the five **SAST** tools: i) 10 two-version combinations (C_2^5); ii) 10 three-version combinations (C_3^5); iii) 5 four-version combinations (C_4^5); and iv) 1 five-version combination (C_5^5). To characterize the performance of the different combinations, we use two conventional statistical measures for the performance of a binary classification systems:

We can classify the decisions of a **SAST** tool into four classes (like for any other binary decision system): False Positive (**FP**) and True Negative (**TN**), regarding code that is not vulnerable, and False Negative (**FN**) and True Positive (**TP**), regarding code that is vulnerable. The conventional statistical measures for the performance of a binary classification test that we have used, *Sensitivity* (recall or **TPR**) and *Specificity* (**TNR**), are illustrated in Figure 5.2 and are summarized next:

- **Sensitivity.** The rate of detecting a vulnerability. Sensitivity measures the performance of the **SAST** tool to find vulnerabilities (equivalent to Recall):

$$Sensitivity = Recall = \frac{TP}{TP + FN} \quad (5.1)$$

- **Specificity.** The rate of remaining silent when no vulnerability exists. Specificity measures the performance of the **SAST** tool to not raise false alarms:

$$Specificity = \frac{TN}{TN + FP} \quad (5.2)$$

Although there are many other measures available, we used these as they fit well to what we intend to measure and are widely used in literature on decision systems. Other measures can be derived either from these or directly from the **FP**, **TN**, **FN** and **TP** counts.

In summary, to do our analysis we performed the following steps for each plugin in the workload:

- 1) We calculated the **FP**, **FN**, **TP**, **TN** counts for each diverse configuration;
- 2) We calculated the measures of interest (*sensitivity* (Equation 5.1) and *specificity* (Equation 5.2)) for each diverse configuration;
- 3) We generated the Receiver Operating Characteristic (**ROC**) plots showing all the diverse configurations and the individual **SAST** tools;
- 4) We calculated the differences in the measures of interest between diverse configurations and individual systems to obtain the possible improvements or deterioration from switching to a diverse system.

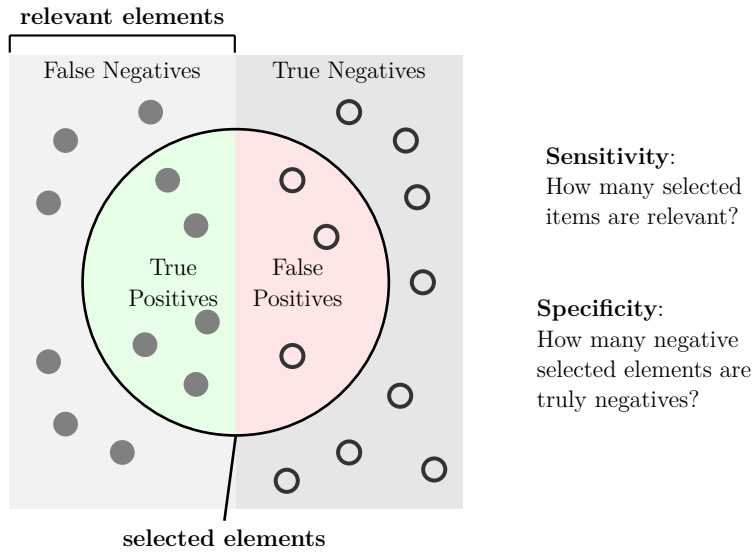


Figure 5.2.: Sensitivity and specificity. Adapted from [198].

5.2.2. Diversity of the Individual SAST Tools

We start by providing information about the performance of the individual **SAST** tools and then we discuss the results of the diversity analysis. Table 5.3 presents the performance of the five **SAST** tools, which can be identified by the labels in round brackets. In the table, we can also see the **FP**, **TN**, **FN** and **TP** counts, respectively, for **SQLi** and **XSS VLOCs**. Table 5.4 presents the *Sensitivity*, and *Specificity* measures for each **SAST** tool. As seen before, for **SQLi** and **XSS VLOCs**, the **SAST** tools reported a very different number of **TPs** and **FPs**, exhibiting different strengths and weaknesses across the classes of vulnerabilities. It is important to emphasize that no **SAST** tool was able to detect all **VLOCs**. In fact, for **SQLi**, the best sensitivity (0.589) is for **phpSAFE** and, for **XSS**, the best sensitivity (0.662) is for **RIPS**. This means that by using the best **SAST** tool for **XSS**, over 40% of the **VLOCs** still remain undetected. Similarly, using the best **SAST** tool for **SQLi** about 30% of the **VLOCs** still remain undetected.

Decision makers also frequently use the **ROC** curves for each system of interest when analyzing the decisions of a binary classifier. **ROC** curves are used to determine how a threshold should be set for a decision system to get an optimal configuration that maximizes the **TP** and minimizes the **FP** rates (what is “optimal” for a give system will inevitably depend on the relative cost that the decision maker assigns to the **FP** and **FN** failures). However, since the systems in our case are already pre-configured, the **ROC** plots show only a point for each system. By showing all the points for the single and diverse systems in the same plot, we can visualize which systems are optimally configured for a given application. Figure 5.3 shows the **ROC** plots for each single **SAST** tool for **SQLi** and **XSS** vulnerabilities.

Table 5.3.: The 5 **SAST** tools and the FP, TN, FN and TP counts by POP and OOP.

SAST tool	SQLi				XSS			
	FP	TN	FN	TP	FP	TN	FN	TP
phpSAFE (A)	72	4,757	265	379	213	15,840	22,93	2,056
RIPS (B)	100	4,729	462	182	454	15,599	14,69	2,880
WAP (C)	9	4,820	490	154	25	16,028	39,64	385
Pixy (D)	12	4,817	580	64	172	15,881	33,13	1,036
WeVerca (E)	35	4,794	605	39	24	16,029	34,88	861

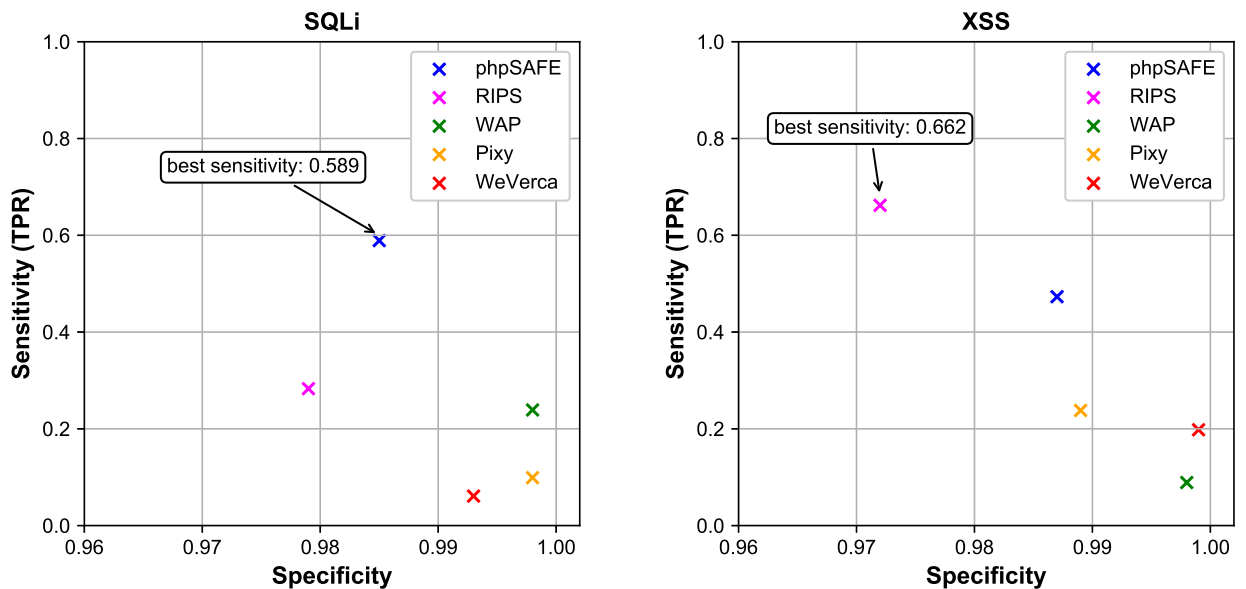
POP								
SAST tool	FP	TN	FN	TP	FP	TN	FN	TP
phpSAFE(A)	12	639	145	61	163	12,316	2,229	1,069
RIPS (B)	100	551	25	181	424	12,055	513	2,785
WAP (C)	0	651	148	58	17	12,462	2,959	339
Pixy (D)	12	639	142	64	162	12,317	2,352	946
WeVerca (E)	35	616	167	39	23	12,456	2,651	647

OOP								
SAST tool	FP	TN	FN	TP	FP	TN	FN	TP
phpSAFE(A)	60	4,118	120	318	50	3,524	64	987
RIPS (B)	0	4,178	437	1	30	3,544	956	95
WAP (C)	9	4,169	342	96	8	3,566	1,005	46
Pixy (D)	0	4,178	438	0	10	3,564	961	90
WeVerca (E)	0	4,178	438	0	1	3,573	837	214

SAST tool Label: phpSAFE (A), RIPS (B), WAP (C), Pixy (D), WeVerca (E).

Table 5.4.: *Sensitivity* and *Specificity* measures for the 5 **SAST** tools.

SAST tool	SQLi		XSS	
	<i>Sensitivity</i>	<i>Specificity</i>	<i>Sensitivity</i>	<i>Specificity</i>
phpSAFE (A)	0.589	0.985	0.473	0.987
RIPS (B)	0.283	0.979	0.662	0.972
WAP (C)	0.239	0.998	0.089	0.998
Pixy (D)	0.099	0.998	0.238	0.989
WeVerca (E)	0.061	0.993	0.198	0.999

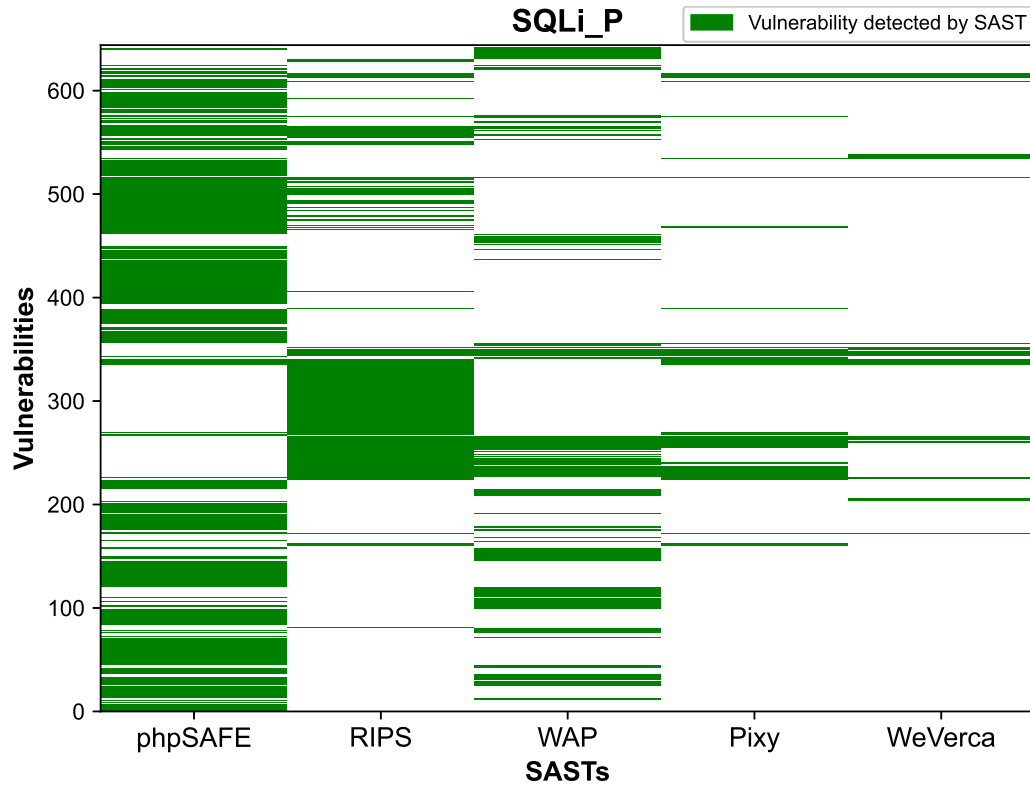
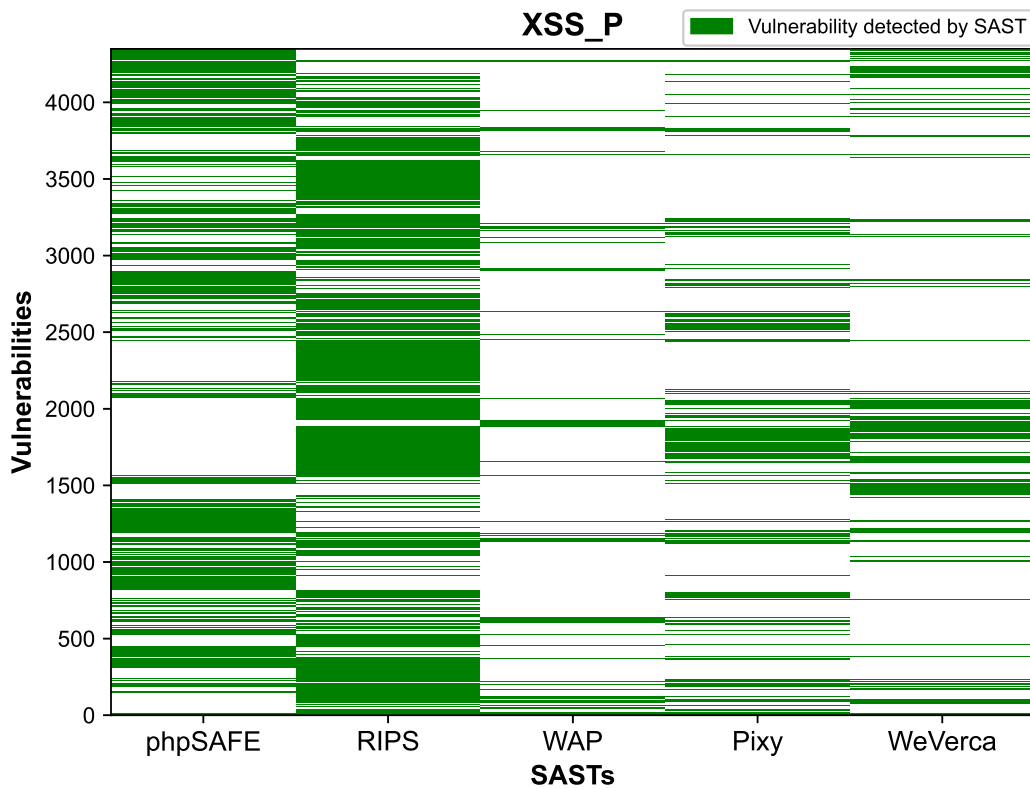


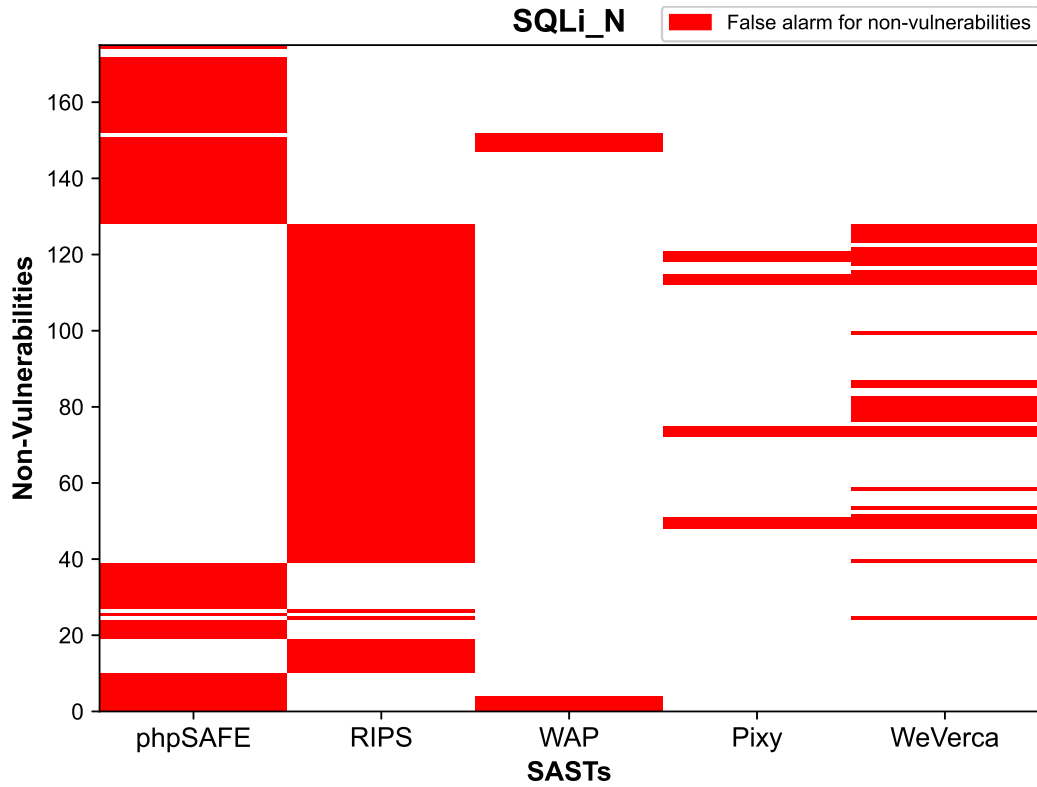
(a) Measures for SQLi and for all plugins.

(b) Measures for XSS and for all plugins.

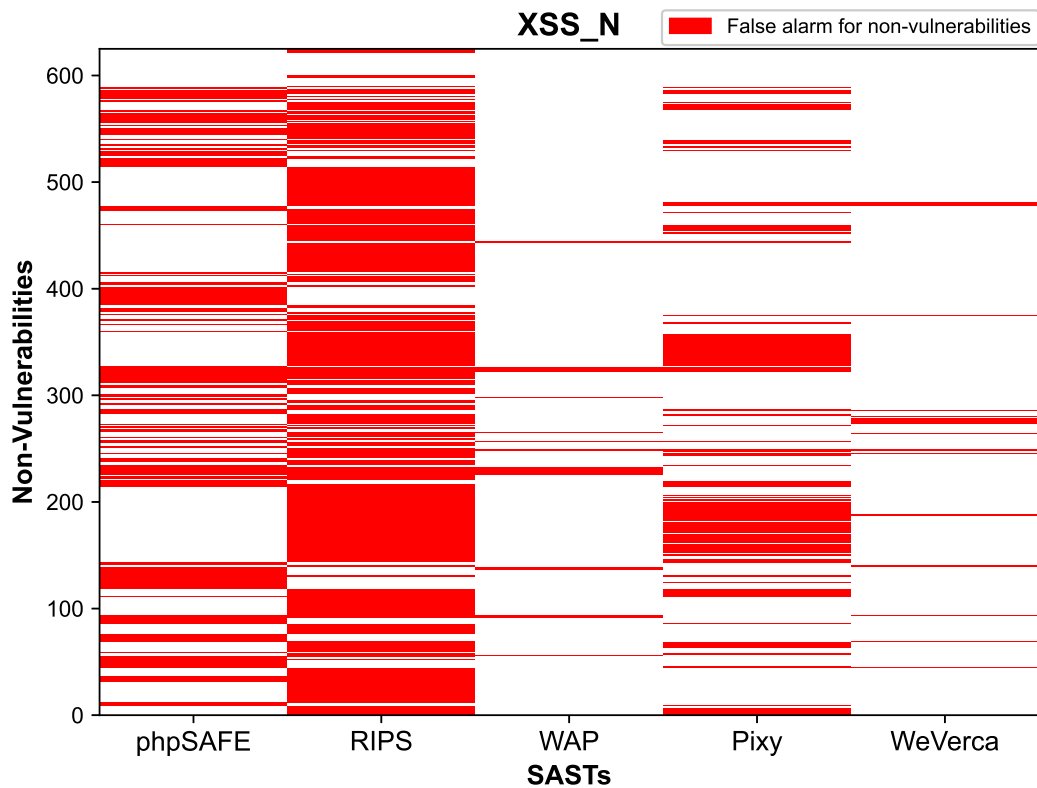
Figure 5.3.: Sensitivity and Specificity measures for each **SAST** tool.

Figure 5.4 and Figure 5.5 show a brief overview of the commonality and diversity of the SAST tools on the vulnerable (VLOCs) and non-vulnerable (NVLOCs) code. The *x-axis* lists the five SAST tools, while the *y-axis* lists the VLOCs (644 for SQLi and 4349 for XSS). A green cell in Figure 5.4 represents, for each SAST tool, whether it detected the vulnerable code as such. In practice, the green cells represent true alarms (TPs) and the white cells represent no alarms (in this case, FNs). Figure 5.5 is similar but, in these plots, we visualize the responses from SAST tools on code that is not vulnerable (i.e., NVLOC). Hence, an alarm is a false positive (FP) represented by a red colored cell and no alarms are again represented as white cells (in this case they are TNs) for the SQLi and XSS VLOCs. Note that, we show only the NVLOCs on which at least one of the SAST tools reports an FP. From the plots presented, we can observe that there is a noticeable diversity between some of the SAST tools. For instance, there is a considerable diversity for both SQLi and XSS between phpSAFE and RIPS, as it is evident by the limited overlap in their alarms. We will elaborate where the diversity is coming from in Section 5.2.5-Identifying Strengths and Weaknesses of SAST Tools.

(a) **SQLi** (VLOC: 644).(b) **XSS** (VLOC: 4,349)Figure 5.4.: Diversity between **SAST** tools for **SQLi** and **XSS** for vulnerable code.



(a) **SQLi** (NVLOC: 4,829, of which 200 in the y-axis below for NVLOCs with FPs for at least one of the SAST tools. The rest had no FPs on any SAST tool).



(b) **XSS** (NVLOC: 16,053, of which 700 in the y-axis below for NVLOCs with FPs for at least one of the SAST tools. The rest had no FPs on any SAST tool).

Figure 5.5.: Diversity between SAST tools for SQLi and XSS for non-vulnerable code.

5.2.3. Results for Diverse SAST tools

We proceeded to calculate the *Sensitivity* and *Specificity* for each of the diverse combinations of the five **SAST** tools, for the three types of adjudication setups considered (namely 1ooN, simple Majority vote (2oo3, 3oo4 and 3oo5) and NooN). Table 5.5 presents the results of this analysis for all the possible two-version, three-version, four-version and five version combinations for **SQLi**. Table 5.6 shows the results for **XSS**.

Table 5.5.: Sensitivity and specificity for the 1ooN, majority vote and NooN configurations for N between 2 and 5 for **SQLi**.

Combination	1ooN		Majority		NooN	
	<i>Sensitivity</i>	<i>Specificity</i>	<i>Sensitivity</i>	<i>Specificity</i>	<i>Sensitivity</i>	<i>Specificity</i>
(A, B)	0.7821	0.9644	-	-	0.0850	1.0000
(A, C)	0.7697	0.9849	-	-	0.0556	0.9983
(A, D)	0.6553	0.9826	-	-	0.0294	1.0000
(A, E)	0.6244	0.9778	-	-	0.0216	1.0000
(B, C)	0.4436	0.9774	-	-	0.0773	1.0000
(B, D)	0.2890	0.9793	-	-	0.0912	0.9975
(B, E)	0.2968	0.9793	-	-	0.0448	0.9928
(C, D)	0.2798	0.9957	-	-	0.0587	1.0000
(C, E)	0.2767	0.9909	-	-	0.0232	1.0000
(D, E)	0.1113	0.9928	-	-	0.0479	0.9975
(A, B, C)	0.9011	0.9642	0.1932	0.9983	0.0124	1.0000
(A, B, D)	0.7867	0.9644	0.1530	0.9975	0.0263	1.0000
(A, B, E)	0.7960	0.9644	0.1113	0.9928	0.0201	1.0000
(A, C, D)	0.7836	0.9824	0.1376	0.9983	0.0031	1.0000
(A, C, E)	0.7867	0.9776	0.0974	0.9983	0.0015	1.0000
(A, D, E)	0.6677	0.9778	0.0556	0.9975	0.0216	1.0000
(B, C, D)	0.4467	0.9774	0.1190	0.9975	0.0541	1.0000
(B, C, E)	0.4575	0.9774	0.1020	0.9928	0.0216	1.0000
(B, D, E)	0.3014	0.9793	0.0943	0.9928	0.0448	0.9975
(C, D, E)	0.2921	0.9909	0.0835	0.9975	0.0232	1.0000
(A, B, C, D)	0.9011	0.9642	0.0866	1.0000	0.0031	1.0000
(A, B, C, E)	0.9134	0.9642	0.0510	1.0000	0.0015	1.0000
(A, B, D, E)	0.7991	0.9644	0.0526	0.9975	0.0201	1.0000
(A, C, D, E)	0.7960	0.9776	0.0448	1.0000	0.0015	1.0000
(B, C, D, E)	0.4590	0.9774	0.0788	0.9975	0.0216	1.0000
(A, B, C, D, E)	0.9134	0.9642	0.0943	0.9975	0.0002	1.0000

SAST tool Label: phpSAFE (A), RIPS (B), WAP (C), Pixy (D), WeVerca (E).

From these tables, we can see some patterns emerging: 1ooN systems are better at finding **VLOCs** (better sensitivity), compared with the best individual **SAST** tools. On the other hand, NooN systems are better at correctly labeling non-vulnerable code (higher specificity). This is to be expected since:

1) **1ooN systems should in all cases perform:**

- *better or equal* than the best single **SAST** tool in the diverse combination N for *vulnerable code*, as any “alarm” from any one of the N **SAST** tools systems leads to an alarm in a 1ooN system;

Table 5.6.: Sensitivity and specificity for the 1ooN, majority vote and NooN configurations for N between 2 and 5 for XSS.

Combination	1ooN		Majority		NooN	
	<i>Sensitivity</i>	<i>Specificity</i>	<i>Sensitivity</i>	<i>Specificity</i>	<i>Sensitivity</i>	<i>Specificity</i>
(A, B)	0.9628	0.9630	-	-	0.1720	0.9955
(A, C)	0.5217	0.9863	-	-	0.0400	0.9989
(A, D)	0.6307	0.9779	-	-	0.0800	0.9981
(A, E)	0.5863	0.9856	-	-	0.0840	0.9996
(B, C)	0.6866	0.9713	-	-	0.0640	0.9988
(B, D)	0.6827	0.9708	-	-	0.2180	0.9902
(B, E)	0.7333	0.9716	-	-	0.1270	0.9986
(C, D)	0.2709	0.9884	-	-	0.0560	0.9994
(C, E)	0.2513	0.9972	-	-	0.0350	0.9998
(D, E)	0.3343	0.9885	-	-	0.1020	0.9993
(A, B, C)	0.9814	0.9628	0.2083	0.9950	0.0340	0.9991
(A, B, D)	0.9669	0.9629	0.3424	0.9859	0.0640	0.9989
(A, B, E)	0.9864	0.9630	0.3095	0.9942	0.0370	0.9998
(A, C, D)	0.6553	0.9777	0.1127	0.9971	0.0320	0.9996
(A, C, E)	0.6132	0.9853	0.1329	0.9985	0.0130	0.9999
(A, D, E)	0.6813	0.9773	0.1888	0.9974	0.0390	0.9998
(B, C, D)	0.7020	0.9706	0.2361	0.9893	0.0510	0.9996
(B, C, E)	0.7510	0.9712	0.1692	0.9977	0.0290	0.9998
(B, D, E)	0.7466	0.9708	0.2571	0.9893	0.0950	0.9994
(C, D, E)	0.3566	0.9877	0.1433	0.9986	0.0250	0.9999
(A, B, C, D)	0.9814	0.9628	0.0883	0.9981	0.0306	0.9997
(A, B, C, E)	0.9984	0.9628	0.0731	0.9988	0.0131	0.9999
(A, B, D, E)	0.9903	0.9629	0.1387	0.9981	0.0320	0.9999
(A, C, D, E)	0.6963	0.9771	0.0711	0.9992	0.0124	1.0000
(B, C, D, E)	0.7595	0.9706	0.1251	0.9989	0.0246	0.9999
(A, B, C, D, E)	0.9984	0.9628	0.1536	0.9975	0.0034	1.0000

SAST tool Label: phpSAFE (A), RIPS (B), WAP (C), Pixy (D), WeVerca (E).

- *equal or worse* than the worst single SAST tool in the diverse combination N for *non-vulnerable code*, as any “alarm” from any single SAST tool leads to this code being incorrectly labeled as vulnerable.

2) NooN systems should in all cases perform:

- *better or equal* than the best single SAST tool for *non-vulnerable code* as the NooN system only raises an “alarm” for *non-vulnerable code* if all the SAST tools in the diverse configuration do so;
- *equal or worse* than the worst single SAST tool system in the diverse configuration N for *vulnerable code*, as the NooN system only labels code as vulnerable if all the single SAST tools in the diverse configuration do so.

Majority voting setups usually balance out these extremes, as they are not as “trigger happy” as 1ooN setups in raising alarms, but are also not as conservative as NooN setups in remaining silent. What is important to understand is how much better, or how much worse, would a diverse configuration performs in these setups, and the results in Tables 5.5 and 5.6 provide us with some interesting observations:

- *Sensitivity*: Combining phpSAFE (A), RIPS (B) and WAP (C) in a 100N setup gives very large gains in sensitivity for both **SQLi** and **XSS**. Sensitivity for the best of these tools for **SQLi** is 0.589. 1003 configuration of these three tools (as listed in the row (A, B, C)) is 0.9. Adding the remaining two **SAST** tools (Pixy and WeVerca) just improves sensitivity a little bit (to 0.901) in a 1005 setup (row (A, B, C, D, E)). For **XSS**, phpSAFE (A) and RIPS (B) in a 1002 setup have a sensitivity score of 0.963 (individually, RIPS (B) has the best sensitivity with 0.662). Combining all five **SAST** tools in a 1005 setup means that almost all the **XSS VLOCs** in the plugins we considered were detected (0.9984). As we would expect, we see large deteriorations in sensitivity for NooN setups. We also observe poor sensitivity results for majority voting setups.
- *Specificity*: We see gains in specificity in NooN setups. Many configurations never raise false alarms in this case. However, they also have very poor sensitivity values. As expected, majority voting setups have better sensitivity compared with NooN, but they also have worse specificity.

Figure 5.6 shows the eight **ROC** plots, one for each class of vulnerability (**SQLi** and **XSS**), and for each configuration of N , $2 \leq N \leq 5$. In addition to the 100N, simple majority (1003, 3004 and 3005), and NooN setups that we presented in tables 5.5 and 5.6, we also calculated the remaining voting setups (2004, 2005, 4005). The most optimal system in a **ROC** plot is the one that appears on the top-right corner (i.e., one that has both sensitivity and specificity of 1, since it detects all **VLOCs** and never raises an alarm for code that does not contain **VLOCs**). We have no such system in our configurations. As we have seen from the results in tables 5.5 and 5.6, most of the results in our configurations have extremes of high sensitivity (100N) or high specificity (NooN). The **ROC** plots make it easier to identify configurations that lie between these extremes by analyzing the spread in the two-dimensional plot.

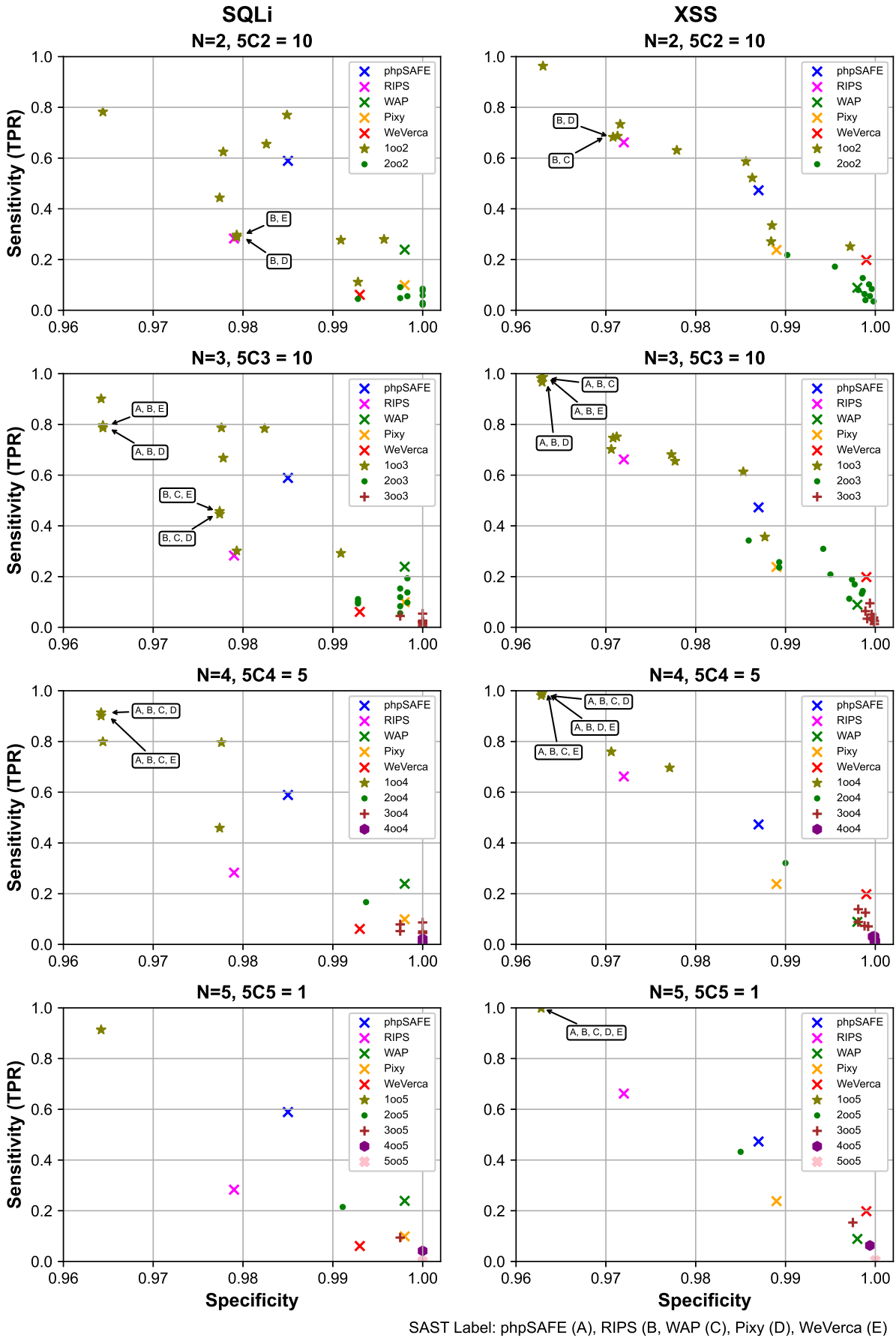


Figure 5.6.: ROC plots for different diverse combinations and the two classes of vulnerabilities.

We conclude our analysis with a summary (Table 5.7) showing the average *Sensitivity* and *Specificity* for non-diverse setups (abbreviated “1v” in the first row of the table) and the averages for the diverse configurations. We provide up to 5 decimal points for **SQLi** and up to 6 decimal points for **XSS** to make the differentiation between high values (multiple nines) and actual 1s (i.e., perfection) as clear as possible. These results confirm the observations we made so far:

- 1) **For 1ooN systems:** we see more than 50% improvements in sensitivity on average in a 1oo2 setup compared with the average individual **SAST** tools. They have also more than three times the improvement in sensitivity on average of a 1oo5 setup compared with individual **SAST** tools. However, this comes at a correspondingly high deterioration in specificity.
- 2) **For NooN systems:** we see that almost perfect specificity can be achieved when using NooN setups (especially for configurations of $N > 2$). But this comes with a large deterioration in sensitivity.
- 3) **Simple majority voting:** these setups on average lead to a deterioration in sensitivity (between 50-65%) but with some improvements in specificity.

Table 5.7.: Average sensitivity and specificity for each diverse version and each class of vulnerabilities. 1v - Average for non-diverse setups.

Diverse Version	SQLi		XSS	
	<i>Sensitivity</i>	<i>Specificity</i>	<i>Sensitivity</i>	<i>Specificity</i>
1v	0.25317	0.99340	0.331938	0.988937
1oo2	0.45286	0.98250	0.566061	0.980060
1oo3	0.62195	0.97559	0.744079	0.972933
1oo4	0.77372	0.96956	0.885169	0.967246
1oo5	0.91345	0.96417	0.998390	0.962811
2oo2	0.05348	0.99861	0.097816	0.997813
3oo3	0.02287	0.99975	0.041711	0.999564
4oo4	0.00958	1.00000	0.022534	0.999875
5oo5	0.00021	1.00000	0.003361	0.999875
2oo3	0.11468	0.99633	0.210025	0.994313
2oo4	0.16662	0.99366	0.320809	0.989996
2oo5	0.21484	0.99110	0.432283	0.984987
3oo5	0.09428	0.99752	0.153599	0.997508
4oo5	0.04173	1.00000	0.063003	0.999377

5.2.4. Testing the Hypotheses

Based on our findings, we conclude that the hypotheses H_1 , H_2 , H_3 , H_4 , H_5 and H_6 , stated in the Section 5.2.1, are not false. Hypothesis H_1 (the sensitivity obtained for combinations of **SAST** tools using *1-out-of-n* adjudication is higher than the sensitivity obtained using *majority* adjudication) and H_2 (the sensitivity obtained for combinations of **SAST** tools using *1-out-of-n* adjudication is higher than the sensitivity obtained using *n-out-of-n* adjudication) are not false because for all combinations with *1-out-of-n* systems (see Tables 5.5 and 5.6) the sensitivity is always better than the sensitivity with *majority* and *n-out-of-n* systems. Hypothesis H_3 (the sensitivity obtained for combinations of **SAST** tools using *majority* adjudication is higher than

the sensitivity obtained using *n-out-of-n* adjudication) are not false because for all combinations with *majority* systems the sensitivity is always better than the sensitivity with *n-out-of-n* systems.

Hypothesis H_4 (the specificity obtained for combinations of SAST tools using *1-out-of-n* adjudication is lower than the specificity obtained using *majority* adjudication) and H_5 (the specificity obtained for combinations of SAST tools using *1-out-of-n* adjudication is lower than the specificity obtained using *n-out-of-n* adjudication) are not false because for all combinations with *1-out-of-n* systems the specificity is always worse than the specificity with *majority* and *n-out-of-n* systems. Hypothesis H_6 (the sensitivity obtained for combinations of SAST tools using *majority* adjudication is higher than the sensitivity obtained using *n-out-of-n* adjudication) are not false because for all combinations with *majority* systems the specificity is always worse than the specificity with *n-out-of-n* systems.

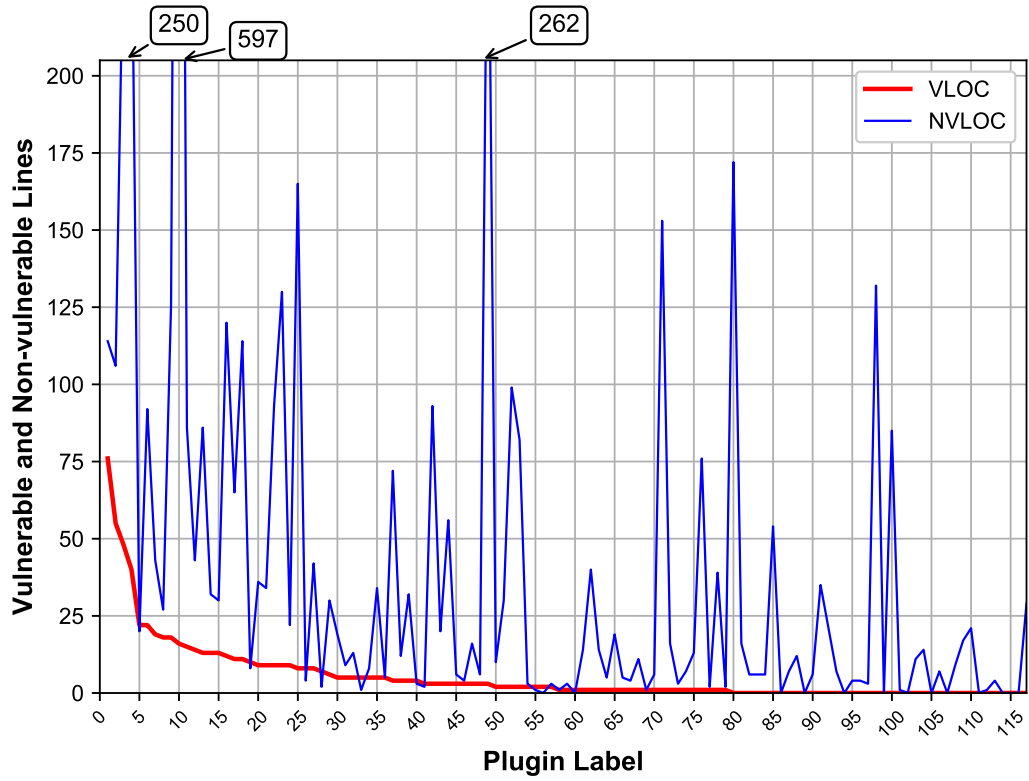
In summary, the main advantage of combining the results of several SAST tools using a *1-out-of-n* strategy is the identification of more vulnerabilities. However, this comes with more FPs with degradation in the specificity. These results show that a *1-out-of-n* strategy should be chosen for all situations where the goal is to find the highest number of vulnerabilities even if reporting many FPs, while a *n-out-of-n* strategy is indicate for situations where resources available for fixing vulnerabilities is very limited, as it leads to few FPs and a higher specificity). However, it also comes with a degradation in the sensitivity leaving many vulnerabilities undetected. The sensitivity for the *Majority* strategy is better than the *n-out-of-n* strategy, making it useful for situations where there are considerable resources for fixing vulnerabilities.

5.2.5. Identifying Strengths and Weaknesses of SAST Tools

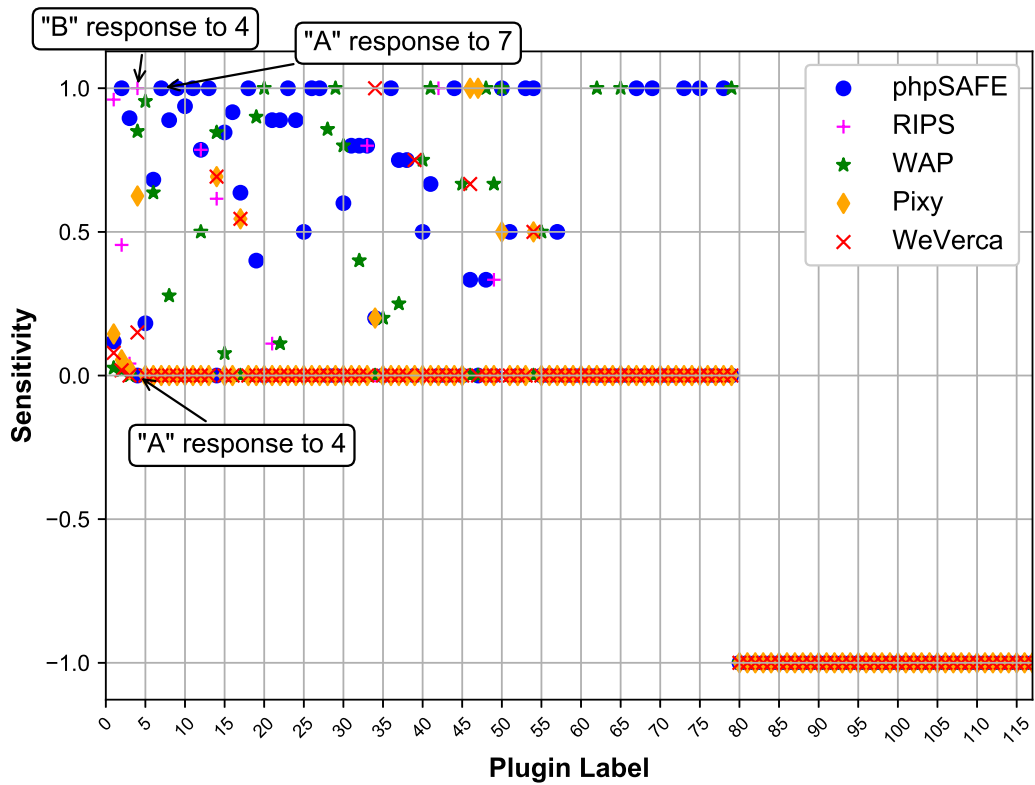
In this section, we present the results of our analysis on the potential diversity of the detection capabilities of the SAST tools. Figures 5.7a and 5.7b show the ordering of the plugins by the total number of VLOCs (those with more VLOCs are on the left side of the graph) for SQLi and XSS, respectively. Figures 5.8a and 5.8b show the sensitivity of each SAST tool for each of the plugins (the order in the x-axis of figures 5.7a and 5.8a corresponds to the order in figures 5.7b and 5.8b, respectively). We see that there is considerable diversity in the sensitivity of the tools for the different plugins. For example, figure 5.8b shows a large cluster of magenta plus (+) (RIPS (B)) in the top left, which indicates that this tool was outperforming phpSAFE (A) on sensitivity for these plugins, even though phpSAFE was better on average overall. For SQLi, RIPS (B) reports many VLOCs in the `levelfourstorefront.8.1.14` plugin (4) and phpSAFE (A) reports none (we highlighted this plugin in Figure 5.8a). However, for the `sendit.2.1.0` plugin (7), SAST tool phpSAFE reports many VLOCs and the other SAST tools report none.

5.2.5.1. In-depth Analysis of the Plugins

As we can see in Figure 5.7, there are SAST tools reporting many VLOCs in some plugins while other SAST tools report few or no VLOCs in those same plugins. In practice, the effectiveness of the SAST tools depends on the types of code constructs that are used. Knowing the reasons of this diversity is an important matter for users and developers. Thus, in this subsection, we

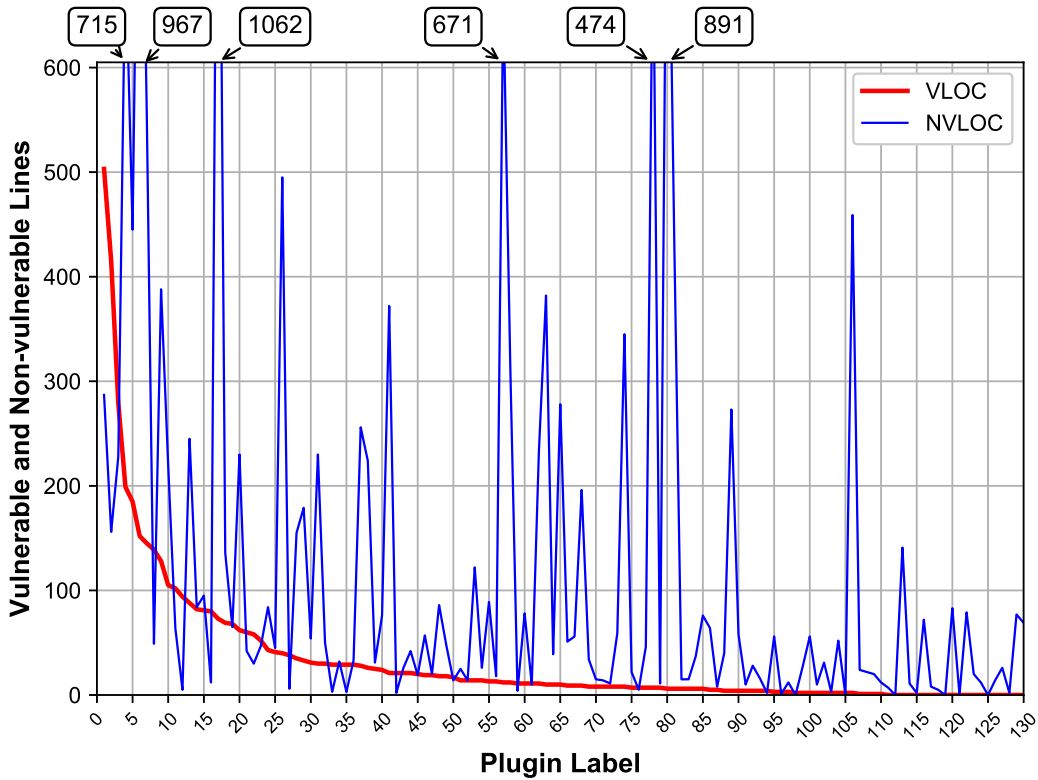


(a) SQLi Plugins.

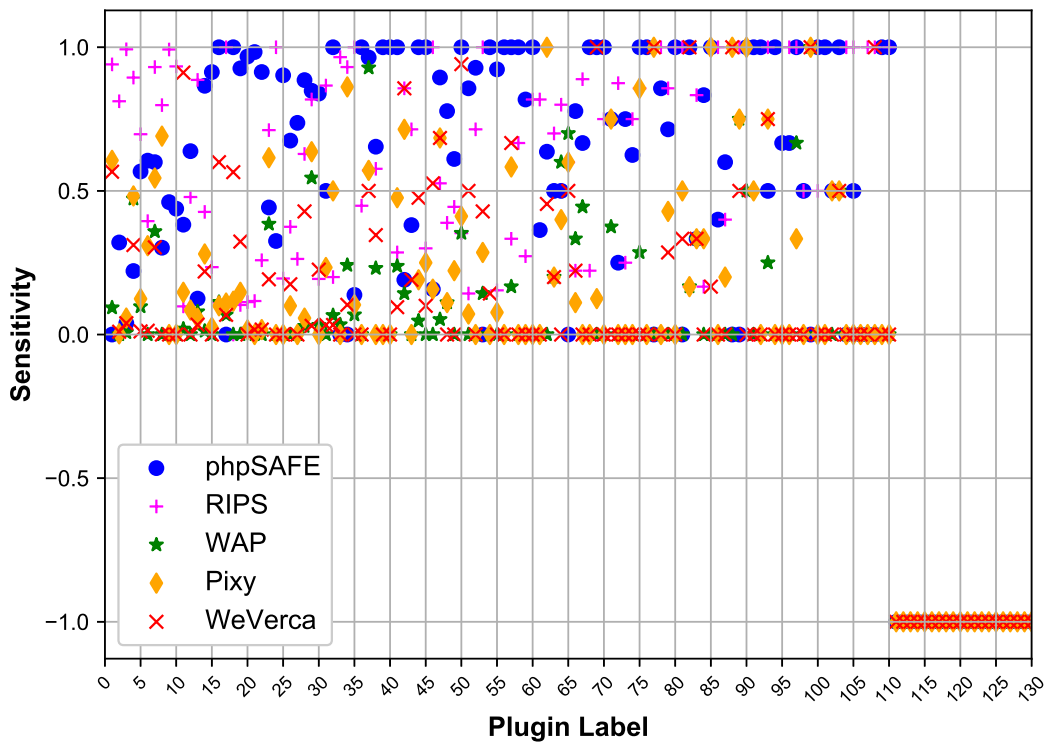


(b) SQLi Sensitivity of *SAST* tool per Plugin.

Figure 5.7.: Plugins *VLOC* and *NVLOC* count and plugins sensitivity for *SQLi*.



(a) XSS Plugins.



(b) XSS Sensitivity of SAST tool per Plugin.

Figure 5.8.: Plugins **VLOC** and **NVLOC** count and plugins sensitivity for **XSS**.

investigate them in more detail. To do so, we use the follow the following steps:

- 1) Sort the plugins in descending order of the number of **VLOCs** in separate lists for **SQLi** (Table 5.8) and for **XSS** (Table 5.9).
- 2) Calculate the number of **VLOCs** reported exclusively by each **SAST** tool and the **VLOCs** reported by several **SAST** tools.
- 3) From each one of these two lists (**SQLi** and **XSS**), select the first 20 plugins (to limit the time required in the process of analyzing the code in-depth) that have at least one **VLOC** detected exclusively by just a single **SAST** tool (i.e., not detected by any one of the other four **SAST** tools).
- 4) For each **VLOC** reported exclusively by just one **SAST** tool, collect the **LOCs** from the **EPs** to the **SS**. This slice of code can be PHP constructs (i.e., if), functions or class methods.
- 5) For each PHP construct collected, define one or more test cases to investigate the reasons why the other four **SAST** tools failed the detection. The test cases may be the complete slice of code or just a subset. In some tests, the code may even be modified (e.g., removing **LOCs**, removing parts of **SQL** statements, swapping **LOCs**) in order to better identify in which conditions the **SAST** tool fails to detect the **VLOC**.
- 6) Use all **SAST** tools to search for **VLOCs** in the test cases.
- 7) Collect and analyze the results of the **SAST** tools to better understand their behaviors.

Table 5.8 and Table 5.9 list these top 20 plugins with **VLOCs** detected by just a single **SAST** tool for **SQLi** and **XSS VLOCs**, respectively. The tables show whether the code is **OOP** or not (i.e., **POP** code), the number of **VLOCs** of the plugin (**VLOC** column) and, for each **SAST**, the number of **VLOCs** it reported exclusively divided by the total number of **VLOCs** that it detected. For example, the first row of Table 5.8 shows that **phpSAFE (A)** was able to detect 9 **VLOCs** out of 76 **VLOCs**, and at least one **VLOC** was detected only by **phpSAFE (A)**, and 8 **VLOCs** that were also detected by at least one other **SAST** tool. These tables also allow us to see more clearly the diversity that exists in the **SAST** tools when analyzing the different plugins, complementing the analysis presented in Figure 5.7 and in the previous section. For example, from Table 5.8 we can see that plugin 4 (**levelfourstorefront .8.1.14**) has 40 **VLOCs**, and **RIPS (B)** was able to detect all of them. Moreover, 6 of these **VLOCs** were exclusively detected by **RIPS (B)**.

This in-depth analysis of the **VLOCs** is helpful for getting a better understanding of where the **VLOCs** are located in the PHP constructs present in the code of the plugins. The process for collecting the code constructs (listed in the tables) is illustrated in Figure 5.9. The figure also includes a running example for **SQLi**. The code constructs are obtained from the **LOCs** of the generated test cases, using the following procedure for each class of vulnerability, implemented in a PHP script:

- 1) First, the **ASTs** for all test cases are generated. The PHP script based on the built-in PHP function `token_get_all`⁴ parses the source code of all test cases, in order to generate their **ASTs**. The PHP `token_get_all` function parses a given source code into PHP language tokens and returns an array of token identifiers. Each individual token identifier is either a single character (e.g., = { ([; . !), or a three element array containing the token index

⁴<https://www.php.net/manual/en/function.token-get-all.php>

Table 5.8.: Top 20 of vulnerable **SQLi** plugins.

Plugin	OOP	VLOC	Vulnerabilities by SAST tool				
			A	B	C	D	E
1 js-appointment.1.5	✓	76	1/9	64/73	0/2	0/11	0/6
2 events-registration.5.44	×	55	29/55	0/25	0/1	0/3	0/1
3 fs-real-estate-plugin.2.06.01	✓	48	42/43	0/2	0/0	0/1	0/0
4 levlfourstorefront.8.1.14	✓	40	0/0	6/40	0/34	0/25	0/6
5 simple-support-ticket-system.1.2	×	22	1/4	0/0	18/21	0/0	0/0
6 wpforum.1.7.8	✓	22	8/15	0/0	7/14	0/0	0/0
7 sendit.2.1.0	✓	19	19/19	0/0	0/0	0/0	0/0
8 odihost-newsletter-plugin	✓	18	13/16	0/0	2/5	0/0	0/0
9 wp-championship.5.8	✓	18	18/18	0/0	0/0	0/0	0/0
10 evarisk.5.1.3.6	✓	16	15/15	0/0	0/0	0/0	0/0
11 slider-image.2.6.8	✓	15	15/15	0/0	0/0	0/0	0/0
12 duplicator.0.5.14	✓	14	1/11	2/11	0/7	0/0	0/0
13 gallery-images.1.0.1	✓	13	13/13	0/0	0/0	0/0	0/0
14 ip-logger.3.0	✓	13	0/0	0/8	2/11	0/9	1/9
15 wp-menu-creator.1.1.7	✓	13	11/11	0/0	1/1	0/0	0/0
16 contus-video-gallery.2.8	✓	12	11/11	0/0	0/0	0/0	0/0
17 collision-testimonials.3.0	✓	11	3/7	0/6	0/0	0/6	0/6
18 ip-blacklist-cloud.3.4	×	11	11/11	0/0	0/0	0/0	0/0
19 forum-server.1.7.1	✓	10	8/8	0/0	2/2	0/0	0/0
20 scormcloud.1.0.6.6	✓	10	1/4	0/0	6/9	0/0	0/0

SAST tools label: phpSAFE (A) RIPS(B) WAP(C) Pixy(D) WeVerca (E).

Table 5.9.: Top 20 of vulnerable **XSS** plugins.

Plugin	OOP	VLOC	Vulnerabilities by SAST tool				
			A	B	C	D	E
1 levlfourstorefront.8.1.14	✓	503	0/0	92/473	0/47	0/305	0/285
2 events-registration.5.44	×	415	73/133	282/337	0/0	0/1	0/4
3 js-appointment.1.5	✓	279	0/11	263/277	0/2	0/16	0/11
4 wp-symposium.14.10	✓	199	0/44	45/178	8/94	0/96	3/62
5 flash-album-gallery.2.55	✓	185	46/105	53/129	5/18	0/23	0/2
6 simple-forum.4.3.0	✓	152	92/92	12/60	0/0	0/47	0/2
7 fbpromotions.1.3.3	✓	145	1/87	30/135	9/52	0/79	0/44
8 ip-blacklist-cloud.3.4		139	27/42	13/111	0/0	0/96	0/0
9 wp-photo-album-plus.5.4.18	✓	128	1/59	69/127	0/0	0/0	0/0
10 gigpress.2.3.8	✓	105	7/46	59/98	0/0	0/0	0/0
11 mtouch-quiz.3.06		102	8/39	0/10	0/2	0/15	63/93
12 sp-client-document-manager.2.4.1	✓	94	49/60	28/45	0/0	0/8	0/0
13 evarisk.5.1.3.6	✓	88	3/11	68/78	4/7	0/5	0/3
14 fs-real-estate-plugin.2.06.01	✓	82	39/71	7/35	0/1	0/23	0/18
15 newstatpress.1.0.4	×	81	60/74	7/19	0/0	0/2	0/0
16 all-video-gallery.1.2	✓	80	26/80	0/9	0/9	0/8	0/48
17 usc-e-shop.1.3.12	✓	73	0/0	63/73	0/5	0/8	0/5
18 adrotate.3.9.4	✓	69	25/69	0/0	0/0	0/8	0/39
19 oqey-gallery.0.4.8	✓	68	37/63	5/7	0/0	0/10	0/22
20 dukapress.2.5.9	✓	62	55/60	2/7	0/1	0/1	0/0

SAST tools label: phpSAFE (A) RIPS(B) WAP(C) Pixy(D).

in element 0, the string content of the original token in element 1 and the line number in element 2.

- 2) The **ASTs** are parsed to extract function calls. The function `token_get_all` only recognizes PHP keywords (e.g., `if`, `for`) and few functions (e.g., `print`, `empty`)⁵. Other functions, identifiers, like classes and function calls, are all parsed into token `T_STRING`. A function call is determined by parsing the three consecutive tokens “`T_STRING`”, “`(`” and “`)`”. The parameters of the function call are between the open “`(`” and close “`)`” parentheses. The PHP script parses function calls as described earlier. We consider only the function calls related with the class of vulnerability under analysis. From this stage results a unique list of functions called in the code.
- 3) The **ASTs** are parsed to extract PHP constructs (e.g., `if`, `echo`, `print`). From this stage results a list of unique PHP constructs.
- 4) Finally, a unique list containing all the PHP constructs and all the functions that are called is created.

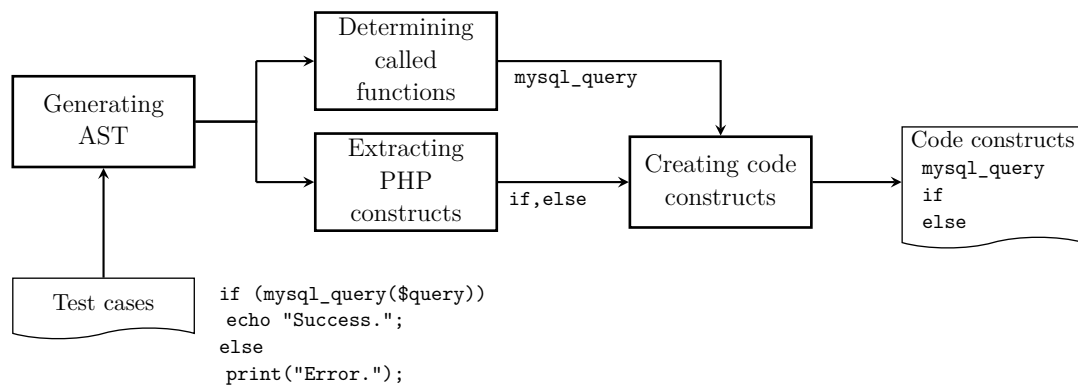


Figure 5.9.: Process for collecting PHP constructs and function calls from test cases.

Table 5.10 and Table 5.11 depict the PHP constructs for which four out of the five **SAST** tools fail to detect the **VLOC**. For each plugin, they show which constructs were successfully parsed by a single **SAST** tool that detected the **VLOC**, but where the other four **SAST** tools failed. To better analyze the diversity, for each plugin, the PHP constructs involved in the same vulnerability are in the same row. Each number in the table represents the number of **VLOCs** detected by the **SAST** tool in that specific construct. For example, in Table 5.10 for the 17th plugin (`collision-testimonials.3.0`), “1A” in the column `mysql_query` and “1A” in the column `if/else` mean that 1 **VLOC** was only detected by `phpSAFE` (abbreviated as tool A, hence “1A” in the table, meaning one **VLOC** detected by tool A) and that the code includes both the `mysql_query` and `if/else` constructs - hence “1A” is noted in both of those columns (see Table 5.10 for the vulnerable source code).

Table 5.10 and Table 5.11 allow us to make two key observations about the context sensitive analysis and the scope of the PHP variables of the code constructs:

- 1) **Context sensitive analysis.** PHP web applications based on the **HTML** user interface may make use of several technologies such **SQL**, **CSS**, **JavaScript**, **jQuery** and **AJAX**.

⁵<https://www.php.net/manual/en/tokens.php>

Table 5.10.: PHP constructs of vulnerable SQLi plugins.

Plugin / Construct		mysql_query	Input func. parameter	if/else	switch	global	foreach	function not executed	query	get_row	get_results	
1	js-appointment.1.5	64B	-	64B					-			
		-	1A						1A			
2	events-registration.5.44	-							1A			
		-							-			
		-							27A			
		1A							-			
3	fs-real-estate-plugin.2.06.01							42A				
4	levelfourstore front.8.1.14			6B				6B				
5	simple-support-ticket-system.1.2			1A			1A		1A		-	
				15C			-		-		15C	
				2C			-		-			
				-			1C		1C			
6	wpforum.1.7.8			6A				6A	6A	-	-	
				-				2A	2A	-	-	
				2C			-	-	-	-	-	2C
				1C			-	-	-	-	1C	-
				2C			-	-	-	-	-	-
				1C			-	-	-	-	-	1C
				1C			1C	-	-	-	-	-
7	sendit.2.1.0			2A		4A	4A	-	-	-		
				1A		15A			15A	1A		
8	odihost-newsletter-plugin			10A	-				10A			
				-	-				2A			
				-	-				1C		1C	
				1C	1C				-			
9	wp-championship.5.8			18A				18A				
10	evarisk.5.1.3.6			15A				15A				
11	slider-image.2.6.8			-					9A			
				6A					6A			
12	duplicator.0.5.14	-	-	1A					1A			
		2B		2B					2B			
13	gallery-images.1.0.1			-					5A			
				8A					8A			
14	ip-logger.3.0	1E		-	1E							
		-		2C	-							
15	wp-menu-creator.1.1.7					1C			-	1C		
						-			11A	11A		
16	contus-video-gallery.2.8				11A				11A			
17	collision-testimonials.3.0	-		2A			2A		2A			
		1A		1A			-		-			
18	ip-blacklist-cloud.3.4			11A					11A			
19	forum-server.1.7.1			-	-			6A	6A			
				-	2A			2A	2A			
				2C	-				2C			
20	scormcloud.1.0.6.6					1A			1A	-		
						6C			-	6C		

SAST tools label: phpSAFE (A) RIPS(B) WAP(C) Pixy(D) WeVerca (E).

Table 5.11.: PHP constructs of vulnerable XSS plugins.

Plugin / Construct	HTML	Javascript context	jQuery context	Input func. par.	if/else	if	ternary operator (?:)	switch	global	foreach (... as ...)	Func. not executed	GLOBALS	Output func. par.	return	mysql_fetch	mysql_fetch_assoc	mysql_fetch_array	->	=>	Associative array []	Index Array	Array elements	explode	print_r	str_replace	User defined function	get_row	get_results	get_var	get_col	echo, die, print			
1	leveledfourstore front.8.1.14				2B	2B									88B																88B 2B			
2	events-registration.5.44														232B	43B																232B 43B 78B 242B		
3	js-appointment.1.5					2C	2C								242B		21B															2C 21B 23B 3C		
4	wp-symposium.14.10				23B		3C													23B												23B 3C 12C 12C		
4	flash-album-gallery.2.55								1A											44A												1A 1A 44A 53B 6B 5C 92A 89A		
6	simple-forum.4.3.0					1B																										1A 1B 6B 1B 4B 1A 1B 18B 9B 1B 5C 4C		
7	fbpromotions.1.3.3				1B							1A								1B												1A 1B 18B 9B 1B 5C 4C		
8	ip-blacklist-cloud.3.4									10A 16A										16A													10A 16A 1A 4B 1B 1B 1B 3B 3B 3B 1A 51B 13B 5B 3A 4A 59B 1A 2A 5A 3E 60E	
9	wp-photo-album-plus.5.4.18					13B																											1A 1A 51B 13B 5B 3A 4A 59B 1A 2A 5A 3E 60E	
10	gigpress.2.3.8				2A																												3A 4A 59B 1A 2A 5A 3E 60E	
11	mtouch-quiz.3.06																																1A 2A 5A 3E 60E	
12	sp-client-document-manager.2.4.1																																15A 4A 28A 2A 6A 1B 1B 6B 6B 3B 3B 11B 11B 3A 3A 63B 5B 4C 32A 1A 4A 3B 1B 2B 2B 1B 60A 26A 1B 1B 5B 1B 10A 35A 1B 2B 1B 12 41A 1B	
13	evarisk.5.1.3.6																																	3A 63B 5B 4C 32A 1A 4A 3B 1B 2B 2B 1B 60A 26A 1B 1B 5B 1B 10A 35A 1B 2B 1B 12 41A 1B
14	fs-real-estate-plugin.2.06.01																																	3A 63B 5B 4C 32A 1A 4A 3B 1B 2B 2B 1B 60A 26A 1B 1B 5B 1B 10A 35A 1B 2B 1B 12 41A 1B
15	newstatpress.1.0.4																																	3A 63B 5B 4C 32A 1A 4A 3B 1B 2B 2B 1B 60A 26A 1B 1B 5B 1B 10A 35A 1B 2B 1B 12 41A 1B
16	allvideo-gallery.1.2																																	3A 63B 5B 4C 32A 1A 4A 3B 1B 2B 2B 1B 60A 26A 1B 1B 5B 1B 10A 35A 1B 2B 1B 12 41A 1B
17	usc-e-shop.1.3.12																																	3A 63B 5B 4C 32A 1A 4A 3B 1B 2B 2B 1B 60A 26A 1B 1B 5B 1B 10A 35A 1B 2B 1B 12 41A 1B
18	adrotate.3.9.4																																	3A 63B 5B 4C 32A 1A 4A 3B 1B 2B 2B 1B 60A 26A 1B 1B 5B 1B 10A 35A 1B 2B 1B 12 41A 1B
19	oqey-gallery.0.4.8																																	3A 63B 5B 4C 32A 1A 4A 3B 1B 2B 2B 1B 60A 26A 1B 1B 5B 1B 10A 35A 1B 2B 1B 12 41A 1B
20	dukapress.2.5.9																																	3A 63B 5B 4C 32A 1A 4A 3B 1B 2B 2B 1B 60A 26A 1B 1B 5B 1B 10A 35A 1B 2B 1B 12 41A 1B

SAST tool label: phpSAFE (A) RIPS(B) WAP(C) Pixy(D) WeVerca (E).

Therefore, the **SAST** tools have to identify these different programming contexts and their relationships to perform the detection of **VLOCs**. In fact, we can observe several code constructs responsible for detection failure that are specific to these technologies. For example, in plugin 12 of Table 5.11, the **VLOCs** of the first three rows are in the context of **HTML** (15A), **Javascript** (4A) and **jQuery** (28A), respectively.

- 2) **PHP scope of the variables**. The scope of a variable is the context within it is defined (e.g., file, function or method). The PHP `global` keyword and the PHP `GLOBALS` array allow the span of the scope of the variables across functions, included classes and required files. Good coding practices do not recommend the use of global variables, because they create dependencies in the codebase and, as the complexity of the codebase grows, these dependencies make it difficult to understand for both humans and tools. This is why the **SAST** tools also struggle to correctly analyze the code in many situations. For example, for the 20th plugin of Table 5.10, the 6 **VLOCs** in columns `global` and `get_row` are due the use of global variables in the PHP code. Additionally, in plugin 11 of Table 5.11, all 60 **VLOCs** in row 5, column `GLOBALS`, are due the use of the `GLOBALS` PHP array. For all these **VLOCs**, the global variables are unsanitized and used in the **SSs**, therefore creating **VLOCs**.

5.2.5.2. Deriving Test Cases from the Plugins

The analysis presented in the previous subsection shows where the **SAST** tools fail in parsing the code when detecting the **VLOCs**. Next, we study why they fail to detect the vulnerable code. To investigate this, we conducted a further study where we mutated the PHP constructs responsible for the failed detection, in a way that the code performed the same function. The idea is to analyze which aspect of the constructs the **SAST** tools are not parsing well, which can be quite helpful for **SAST** tool developers for future improvements, for example.

From our observations, the plugins have 331 **SQLi** and 1739 **XSS VLOCs** reported exclusively by one **SAST** tool. To investigate the reasons why the other **SAST** tools fail the detection of these **VLOCs** we may need at least the same number of test cases. However, there are many **VLOCs** using the same constructs, so there is room for optimization of the experiments. For example, in the 2nd row of Table 5.11, the value 232B on the column `mysql_fetch_assoc` represents 232 **XSS VLOCs** reported exclusively by RIPS. Therefore, instead of creating 232 similar test cases, we created only one test case that can be used to represent all these situations. We followed the same approach for other situations where the order of the constructs does not matter. To systematize the selection order of the plugins and files for the creation of the test cases, we performed the following procedure:

- 1) For each plugin, we sorted the results of the **SAST** tool by the number of lines of code of the vulnerable files;
- 2) Then, we chose the first row for collecting the slice of code for use in the tests.

Therefore, we started with the smallest files because it helps with readability and understanding what the code is doing. Our goal is to create simple test cases maintaining the data/control flows of the original code.

```

1 ...
2 1340 $date = $_REQUEST["date"]; // EP
3 1341 if ($date) { // if construct
4 1342     $where1 = "WHERE date = '$date'";
5 1343 }
6 1345 $tbl_name = "wp_resservation_disp";
7 1347 $adjacents = 3;
8 1353 $query = "SELECT COUNT(*) as num FROM $tbl_name $where1";
9 1355 $total_pages = mysql_fetch_array(mysql_query($query));
10 1356 $total_pages = $total_pages[num] ;
11 1359 $targetpage = "adminphp?page=allbooking";
12 1360 $limit = 10;
13 1361 $pagelist = $_GET['pagelist']; // EP
14 1362 if($pagelist) // if/else construct
15 1363     $start = ($pagelist - 1) * $limit;
16 1364 else
17 1365     $start = 0;
18 1368 $sql = "SELECT id, date,time_start,time_end,max,price,status,description FROM $tbl_name $where1 LIMIT $start,
    $limit";
19 1369* $result = mysql_query($sql); // SS
20 ...

```

Figure 5.10.: Slice of code from `js-appointment.1.5/js-event.php` of the plugin, without blank and commented lines. Target SS in line 1369*.

The generated test cases are based on the results of the **SAST** tools. First, we consider all **LOCs** containing data/control flows beginning at one (or more) **EP** and ending at a target **SS**. The slice of code in Figure 5.10 shows the **LOC** between the **EPs** (i.e., used in the **SS**) and the target **SS** at line 1369 with the vulnerable variable `$where1`. Next, we discard all **LOCs** that are not in the data/control flows of the variables (vulnerable or not) used in the **SS**. According to this, the lines 1347, 1353, 1355, 1356 and 1359 are discarded. Therefore, we maintain the **LOC** required to assign values to the variables used in the **SS** (`$tbl_name`, `$where1`, `$start` and `$limit`). Finally, we add the necessary code (e.g., the “}” symbol to close a `if` block statements (though this is not required in this example as the code already has the “}” in line 1343) to maintain the code syntactically correct and making the **SA** easier to perform.

By following this procedure, we collected 13 slices of code for **SQLi** and 35 for **XSS** as tests for the **SAST** tools. Since the order of the statements in some constructs (e.g. `if/else`, `switch`, `global` etc.) may affects the behavior of the **SAST** tools, we derived several more test cases by swapping the order of the constructs and modifying the code (e.g., negate the condition of an `if` construct) to keep the code equivalent. Overall, we obtained 17 (13+4) test cases for **SQLi** and 39 (35+4) for **XSS**. To clarify this procedure, the next paragraphs provide an example showing how this was done.

Figure 5.11 lists an example of a slice of code from the file `testimonials.php` of the plugin `collision-testimonials.3.0`. To improve reading, we removed blank lines and comments. There is an **SQLi VLOCs** in line 333 due to untrusted data coming from the PHP `GET` array, reaching the **SS** `mysql_query` without sanitization. The variable `$query` is vulnerable because it is built from the concatenation of the unsanitized vulnerable variables `$sort` and `$dir` (i.e., two **EPs** from lines 318 and 326). Therefore, between the **EPs** and the **SS**, there are four possible execution paths (see Figure 5.12 with the data/control flow graph) due to `if/else` control flow

constructs (starting at lines 317 and 325).

To create the test cases, we removed the initialization of the `$sort` and `$dir` variables (lines 316 and 324), which are not necessary because the variables always take a value either in the `if` or in the `else` construct. Also, since the block of code between lines 317 and 322 is similar to the code between lines 325 and 330, we can remove one of them. After these modifications, the resulting test case is listed in Figure 5.13. The test case of Figure 5.14 was obtained by interchanging the statements of the `if/else` construct and the negation of the `if` condition, so it still performs the same operation.

```

1 316 $sort = "";
2 317 if(isset($_GET['sort'])&&!empty($_GET['sort'])){           // if/else construct
3 318     $sort = $_GET['sort'];                                 // EP
4 319 }
5 320 else {
6 321     $sort = "id";
7 322 }
8 323 $dir = "";
9 324 if(isset($_GET['dir'])&&!empty($_GET['dir'])){           // if/else construct
10 325     $dir = $_GET['dir'];                                 // EP
11 326 }
12 327 else {
13 328     $dir = "asc";
14 329 }
15 330 $query = "SELECT * FROM $testimonials ORDER BY $sort $dir";
16 331 $results = mysql_query($query);                         // SS

```

Figure 5.11.: Slice of code from the file `testimonials.php` of the `collision-testimonials.3.0` plugin (row 17 of Table 5.10). It represents the vulnerability **1A** in the PHP constructs `if/else` and `mysql_query` in Table 5.10.

The results of running the SAST tools for the original plugin code (Figure 5.10), and the test cases of Figure 5.11, Figure 5.12 and Figure 5.13 are summarized in Table 5.12. `phpSAFE` and `WAP` correctly report a **VLOC** when the vulnerable variable takes untrusted data in the first part of the `if` construct and they miss the detection of the **VLOC** when the vulnerable variable takes untrusted data in the `else` part of the `if` construct. This means that these SAST tools are likely to miss detection of **VLOCs** that go through the `else` part of `if/else` construct. Moreover, the fact that these SAST tools are not taking into consideration the second part of the `if/else` constructs also highlights where their developers should look at in order to fix them.

Table 5.12.: Results of the SAST for `if/else` Blocks of Code.

Test case	SAST tool				
	phpSAFE	RIPS	WAP	Pixy	WeVerca
Test case original code (Figure 5.10)	×	×	✓	×	×
Test case of Figure of 5.11 (modified version of Figure 5.10)	×	×	✓	×	×
Test case of Figure of 5.13	×	×	✓	×	×
Test case of Figure of 5.14 (modified version of Figure 5.13)	✓	✓	✓	×	×

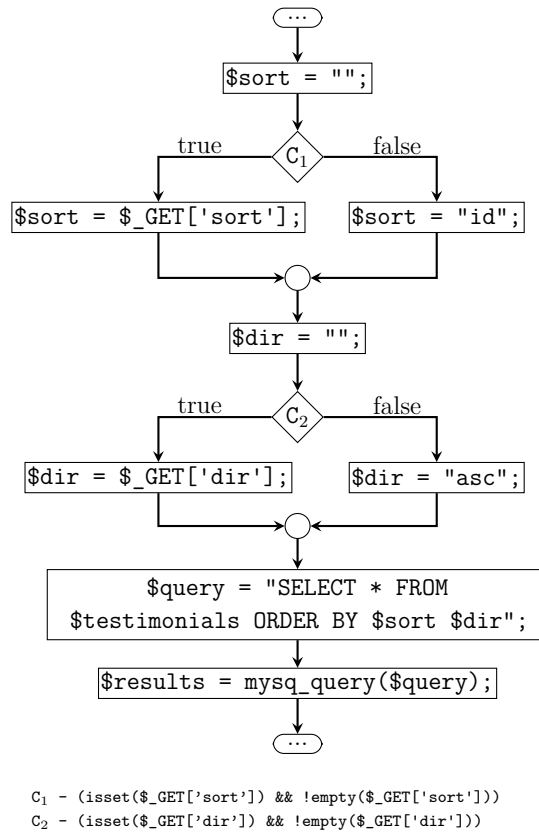


Figure 5.12.: Data flow and control-flow graph of the test case in Figure 5.11.

```

1 if(isset($_GET['sort'])&&!empty($_GET['sort'])){
2     $sort = $_GET['sort'];
3 } else {
4     $sort = "id";
5 }
6 $query = "SELECT * FROM testimonials ORDER BY $sort";
7 $results = mysql_query($query);
  
```

Figure 5.13.: Test case derived from listing in Figure 5.11 after the initial cleaning.

```

1 if(!isset($_GET['sort'])||empty($_GET['sort'])){
2     $sort = "id";
3 } else {
4     $sort = $_GET['sort'];
5 }
6 $query = "SELECT * FROM testimonials ORDER BY $sort";
7 $results = mysql_query($query);
  
```

Figure 5.14.: If/else test case derived from Figure 5.13 by interchanging the statements in the if/else construct and negating the if condition.

5.2.5.3. Evaluation of the SAST Tools in the Test Cases

The results of evaluating the SAST tools with the test cases derived from the slices of code of the vulnerable plugins are depicted in Table 5.13 and Table 5.14, for XSS and SQLi, respectively. In these tables, the symbol $\checkmark \times$ means that we performed two or more test cases. In some of these test cases, the SAST tool misses the VLOCs while it reports the VLOC in the modified test cases. On the other hand, the symbol \checkmark means that it detects the VLOC in all cases we tested. Finally, \times means it failed to detect the VLOCs in all cases tested.

We found several reasons why the SAST tools may fail to detect VLOCs. The main ones are summarized next:

Table 5.13.: Strengths and Weaknesses of SASTs (XSS).

PHP construct	SAST tool				
	phpSAFE	RIPS	WAP	Pixy	WeVerca
Javascript context / HTML	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
jQuery context	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
Input function parameter	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
if/else	$\checkmark \times$	\checkmark	$\checkmark \times$	\checkmark	\checkmark
if	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
ternary operator (?:)	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
switch	$\checkmark \times$	\times	$\checkmark \times$	\checkmark	\checkmark
global	\times	$\checkmark \times$	\checkmark	\checkmark	\checkmark
foreach (\$arr as \$v)	\checkmark	\checkmark	\times	\checkmark	\checkmark
- > (foreach (\$obj as \$p))	\checkmark	\times	\times	\times	\times
=> (foreach(\$a as \$k=>\$v))	\times	\checkmark	\times	\checkmark	\checkmark
function (not executed)	\checkmark	\checkmark	\times	\times	\times
GLOBALS PHP array	\times	\checkmark	\checkmark	\checkmark	\checkmark
Output function parameter (&)	\times	\times	\times	\checkmark	\checkmark
return	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
mysql_fetch_assoc (elements)	\times	\checkmark	\times	\times	\checkmark
mysql_fetch_array (elements)	\times	\checkmark	\times	\times	\checkmark
Operator []	\times	\checkmark	\checkmark	\times	\times
Associative array (\$arr['id'])	\times	\checkmark	\times	\times	\checkmark
Index array (\$arr[0])	\times	\checkmark	\times	\times	\checkmark
Array elements (EP as array)	\times	\checkmark	\times	\times	\checkmark
explode	\checkmark	\checkmark	\checkmark	\checkmark	\times
print_r	\checkmark	\checkmark	\times	\checkmark	\times
str_replace	\checkmark	\checkmark	\times	\checkmark	\checkmark
User function sf_esc_str	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
get_row	\checkmark	\checkmark^1	\checkmark	\times	\times
get_results	\checkmark	\checkmark^1	\checkmark	\times	\times
get_var	\checkmark	\checkmark^1	\checkmark	\times	\times
get_col	\checkmark	\checkmark^1	\checkmark	\times	\times
\$wpdb WP object as EP	\checkmark	\times	\checkmark^1	\times	\times
stripslashes	\checkmark	\checkmark	\times	\checkmark	\checkmark
mysql_real_escape_string	\checkmark	\checkmark	\checkmark	\checkmark	\times
sanitize_text_field	\checkmark	\checkmark	\checkmark	\times	\times
mysqli_query	\checkmark	\checkmark	\checkmark	\times	\times

\checkmark - Success, \times - Fails, \checkmark^1 - SS added to the SAST tools' config file.

Table 5.14.: Strengths and Weaknesses of **SAST**s (SQLi).

PHP construct	SAST tool				
	phpSAFE	RIPS	WAP	Pixy	WeVerca
mysql_query	✓	✓	✓	✓	✓
Input function parameter	✓	✓	✓	×	✓
if/else	✓×	✓	✓×	✓	✓
global	×	✓×	✓	✓	✓
foreach(<i>arr</i> ; <i>sv</i>)	✓	✓	×	✓	✓
function (not executed)	✓	✓	×	×	×
query	✓	✓ ₁	✓	×	×
get_row	✓	✓ ₁	✓	×	×
get_results	✓	✓ ₁	✓	×	×
get_var	✓	✓ ₁	✓	×	×
switch	✓×	✓	✓×	✓	✓
mysql_real_escape_string	✓×	✓	×	✓	×
sanitize_text_field	×	✓	✓	✓	✓

✓ - Success, × - Fails, ✓₁ - **SS** added to the **SAST** tools' config file.

- 1) **Setting up the SAST tool.** Providing a list, by class of vulnerability, of **EP**s, **SS**, sanitization functions, and revert functions (functions that override the protection mechanisms) used in the software to be analyzed is fundamental to correctly configure the **SAST** tool. Therefore, a **SAST** tool with wrong settings may miss **VLOC**s and may produce **FP**s. For example, by adding the **WAP** database of **SS**s (e.g., `query`, `get_row`, `get_results`, `get_var`) to the **RIPS** configuration file (`sinks.php`), we make it able to report previously missed **VLOC**s. This configuration of the specificities of the environment should be a requirement when configuring the **SAST** tool for a new framework, for example.
- 2) **PHP control flow constructs.** There are some **SAST** tools (**phpSAFE** and **WAP**) that miss **VLOC**s in code using simple **PHP** control flow constructs (e.g., `if/else`, `if`, ternary operator(`condition ? val1 : val2`), `switch`, `return`) that may be easily fixed in a future release of the tools.
- 3) **Analyzing functions not called in the code.** We found that some **SAST** tools (**WAP**, **Pixy** and **WeVerca**) do not analyze the code of custom functions that are never called from within the code being analyzed. However, software projects tend to use code from different sources that must work together, like plugins. For example, **WordPress** uses the concept of **hooks**⁶ that allows plugins to register callback functions that should be called by the **WordPress** core when certain specific events happen during the generation of web pages. This is a very important aspect of **WordPress**, since it allows plugins to modify or add features without having to change the **WordPress** core files. Since **SAST** tools are currently used in all phases of the **SDLC**, their capability to analyze code that may be called from elsewhere is of upmost importance because the calling software may not yet be written or integrated.
- 4) **Modeling built-in PHP functions.** The **PHP** built-in functions require a precise modeling of all parameters (i.e., input and output) and function returns. **RIPS** simulates several hundreds of **PHP** built-in features (e.g., `preg_match_all` function) allowing it to precisely

⁶<https://developer.wordpress.org/plugins/hooks>

model the highly dynamic PHP language. This information may increase the number of **VLOCs** found and reduce the number of **FPs** reported. This shows the importance of a thorough configuration of the **SAST** tools that new releases should update.

- 5) **Modeling arrays:** Some **SAST** tools (**phpSAFE**, **WAP** and **Pixy**) have serious limitations dealing with arrays. Since arrays are commonly used in PHP applications, these **SAST** tools miss many **VLOCs**. This is another important indication of need for improvement for the developers of these **SAST** tools.
- 6) **Output format.** The output format of the **SAST** tool should be harmonized in order to ease the automation of the analysis and to be able merge the results of several **SAST** tools. For example, in a single plugin, we found over 15 **VLOCs** where a **SS** occupied several **LOCs** outputting individual array elements or arrays contained in fields of objects. For these cases, we found that the **SAST** tools report the **VLOCs** in different **LOCs**, which makes the comparison of **SAST** tools somewhat cumbersome.
- 7) **Code complexity.** Some **SAST** tools report **VLOCs** in simple test cases, but fail to analyze the same code when inserted in a larger application (i.e., real code). There are several reasons for this behavior, such as a very complex code with many possible paths, or a very large codebase, since a linear increase in number of **LOCs** creates an exponential increase in complexity. Therefore, the **SAST** tools need improvements in order to be able to process these more complex codebases, taking into consideration CPU and memory constraints.

5.3. Threats to Validity

There are several limitations related to the dataset, the **SAST** tools used and the analysis procedure which may impact the main conclusions drawn:

- 1) **Dataset.** There are limitations regarding the scope of the dataset in this experiment, since it considers only WordPress plugins. But, as we stated previously, WordPress is a widely used **CMS** application.
- 2) **Vulnerabilities.** In this study we have considered two classes of vulnerabilities only (**SQLi** and **XSS** only). However, **SQLi** and **XSS** are some of the most widely published threats to web applications.
- 3) **VLOC and NVLOC counts.** Another limitation stems from our classification of **LOCs** in **VLOC** and **NVLOC**. As we stated in Section 4.1.3.3, the list of **VLOCs** in our study are the **TPs** reported by the tools and the vulnerabilities of the **WPVD**. The list of **NVLOCs** is obtained from all **LOCs** with a **SS** with at least one variable, excluding those that were reported by the tools and confirmed manually as **TP**. Even if great care was taken in labeling the **LOCs** in the two categories, this is a best-effort attempt at finding the **VLOCs** and **NVLOCs** in the application.
- 4) **Free SAST tools.** All **SAST** tools used in this study are free. Some of them, such as **Pixy** and **RIPS**, have not been updated for a while. On the other hand, **WAP**, **phpSAFE**, and **WeVerca** are recent tools that can also analyse **OOP** code. There are several other

commercial and free [SAST](#) tools available in the market, which a future work could include in further analysis.

- 5) **Tools configuration.** The dataset used in this study was collected with all tools configured by default for PHP entry points, sensitive sinks and sanitization functions. The results of the tools may be improved (+[TP](#) and -[FP](#)) by adjusting their configuration settings for WordPress built-in database functions, sanitization and escaping routines.
- 6) **Language domains.** The dataset and the tools are for the PHP language. Our choice was deliberate because PHP powers over 79% of web applications [22]. Future work could consider applications written in other languages such as ASP.NET, Java, JavaScript and Python.

5.4. Conclusion

This chapter presented results of analyzing the performance of diverse [SAST](#)s tools configurations. The analysis was performed using the dataset from Chapter 4.2-Benchmark Instantiation, where five [SAST](#) tools were used for finding two types of vulnerabilities, [SQLi](#) and [XSS](#), in 134 WordPress plugins. We presented two case studies.

In the first case study, we addressed the problem of combining the output of several [SAST](#) tools searching for [SQLi](#) and [XSS](#) vulnerabilities in WordPress plugins using *1-out-of-n* strategy. Results show that different combinations of tools achieve different performance, both in terms of vulnerabilities detected and [FP](#)s. Also, there are trade-offs that should be considered when combining several tools, which may affect the selection decision, depending on the scenario where the tools are to be used. We must emphasize that there are even cases where using a single tool provides better results than combining multiple tools. Combining the outputs of several free [SAST](#) tools does not always improve the vulnerability detection performance. Thus, the best solution can be a single [SAST](#) tool or a combination of [SAST](#) tools that may not include all the [SAST](#) tools under evaluation.

In the second case study, we conducted an empirical study looking at all the possible *1-out-of-n*, *n-out-of-n* and *majority voting* adjudications. As three out of the five [SAST](#) tools used for creating the dataset were not able to analyzing several files, we considered only the results obtained for the files that could be successfully analyzed by all five tools. From the five individual [SAST](#) tool, we built 10 diverse pairs, 10 diverse triplets, 5 diverse quadruples and one diverse quintet [SAST](#) tools system. We presented the results using the well-established sensitivity and specificity metrics. The main conclusions are as follows:

- **For 1-out-of-n systems:** improvements in sensitivity compared with individual [SAST](#) tool are from 50% on average for 1-out-of-2 systems, to more than 300% for 1-out-of-5 systems, but they come with a corresponding specificity deterioration. The largest improvements in sensitivity with the least deterioration in specificity are from combining `phpSAFE` with `WAP` [SAST](#) tools in a diverse 1-out-of-2 configuration.
- **For n-out-of-n systems:** specificity can be perfect in most setups, but with severe deterioration in sensitivity on average.

- **For majority voting setups:** average deterioration in sensitivity (between 50% and 65%) but with some improvements in specificity.

For organizations primarily interested in detecting vulnerabilities (improved sensitivity) and that are willing to invest resources in sifting through alarms to separate out the false alarms from true alarms, diverse setups in a *1-out-of-n* adjudication setup can be very beneficial. In particular, phpSAFE, RIPS and WAP SAST tools exhibit considerable diversity in vulnerability detection. In fact, from the analysis of the code of the plugins we identified sources of diversity in the design and configuration of these SAST tools which lead to their diversity in behaviour. For example, some of the SAST tools are better at detecting vulnerabilities in certain code constructs than others (for example, the way in which they analyze arrays, control flow constructs, etc.), which leads to the observed benefits in overall vulnerability detection.

There are several provisions for further work. We plan to automate the process of extracting the slices of code from the plugins and derive test cases. In fact, we found that using small test cases derived from the plugins is one helpful way to find strengths and weaknesses of the SASTs tools. We also plan to investigate optimal adjudication setups that allow us to improve both the sensitivity and specificity depending on types of code that is inspected by these tools. Optimal adjudicators are known to perform much better than conventional *1-out-of-n*, majority or *n-out-of-n* setups.

Blending Static and Dynamic Analysis for Vulnerability Detection

One of the most effective ways to lower the number of vulnerabilities present in the source code during the development lifecycle of an application, is to first conduct a [SA](#) and later on to run [DA](#) tests [199] [200]. The ability to analyze an application both statically and dynamically is of utmost importance, as some vulnerabilities are better detected with [SA](#), while others are with [DA](#). In general, [SA](#) is able to cover 100% of the source code, but it produces many [FP](#) cases. On the other hand, [DA](#) is usually only able to cover part of the code, but it is very precise when it signals a vulnerability [83]. Testing with both [SA](#) and [DA](#) yields the most comprehensive testing and it is considered a must have by secure software development lifecycles. They are, however, typically applied separately and in different stages of the development process: [SA](#) is used earlier when compiling the various software modules and [DA](#) later when the modules are already developed and integrated with each other.

The complementary advantages of [SA](#) and [DA](#) have led researchers into combining them to achieve the best of both worlds (e.g., [129], [138], [83], [137], [139], [149]). One direction to increase the precision of the [SA](#) is to confirm the reported vulnerabilities by using [DA](#) [83]. Some approaches aim at preventing online attacks, thus requiring runtime components of the application while it is in production [201]. Other approaches require the generation of specific test inputs to detect vulnerabilities [202]. The generation of effective test inputs is a hard task and there are research works dedicated to address this problem [203] [142]. Despite the fact that [SA](#) provides the location of the vulnerability (i.e., vulnerable input variable, file name, activation location), it is generally blinded towards the values of other relevant input variables that are going to define the path constraints that must be satisfied so that the vulnerable input variable is able to travel through the code and succeeds in activating the vulnerability [142]. Therefore, approaches based only on the results of [SA](#) to feed the [DA](#) have the limitation of not providing enough data to exploit all the vulnerabilities discovered. In fact, only a small subset of them usually succeed.

This chapter proposes a methodology to effectively increase the number of reported **TPs** while reducing the number of **FPS**, when searching for vulnerabilities using **SA** and **DA**. Our methodology combines **SA** and **DA** with the inclusion of a runtime procedure (crawling) in between, in a way that the information provided by the crawling to the **DA** allows the creation of tailored attack vectors that can effectively be used to increase the number of confirmed **TPs** and, at the same time, reduce the overall number of **FPS**.

In short, the methodology starts by performing a **SA** to produce a list of candidate vulnerabilities. Next, the application is executed, automatically stopping only when the code where the vulnerabilities are located is run. Afterwards, while the application is being executed, the runtime information include the execution of the code where the vulnerabilities are located is gathered. A set of specific inputs and configuration options is automatically generated from the results of the **SA** and the runtime information collected, which will guide the **DA** in the process of successfully exploiting each vulnerability reported by the **SA**. Unlike other approaches, our methodology has the ability to automatically generate the necessary data to feed the **DA**, so it is able to go through the execution path where the vulnerability is located and trigger it, therefore increasing the number of vulnerabilities that can be confirmed by the **DA**.

The approach was experimentally evaluated for **SQLi** vulnerabilities, using the results of a diverse set of **SAST** tools (that overall improve the detection capability of each individual **SAST** tool (see Section 4.2 for more details) that was used to search **SQLi** vulnerabilities in 49 freely available vulnerable WordPress plugins. Results show that our approach is able to confirm 480% more **TPs** and correctly identify 700% more **FPS** than when directly feeding the results of the **SA** to the **DA**. Our approach provides a huge improvement, if we consider developers and security practitioners that need to use the best possible data in order to identify and work on the fixes of the vulnerabilities found by the tools, while spending the least possible amount of resources dealing with **FPS**.

The outline of this chapter is as follows. Section 6.1 details the proposed approach for blending static and dynamic analysis. Section 6.2, describes an instantiation using PHP language and WordPress plugins as workload. Section 6.3 presents an discusses the experimental results. Section 6.4 concludes the chapter.

6.1. Approach for Blending Static and Dynamic Analysis

Our approach for blending static and dynamic analysis is based on the high vulnerability detection rate that **SA** is able to provide and on the use of **DA** to confirm if the vulnerabilities reported are indeed true positives. The **SA** can be done automatically by means of **SAST** tools, like **RIPS** [204] or **WAP** [103], and the **DA** by existing automatic penetration testing tools, like **SQLMap** (for **SQLi**) or **Xsser** (for **XSS**). In practice, for each vulnerability reported by the **SAST** tool (we refer to it as a single tool, but in practice a set of tools can be used, as we will discuss later on), we need to build a custom configuration setup for the execution of the **DA** tool, containing the **URL** of the vulnerable web page (that may not be directly accessible via a **URL** link), the input parameters and their values, the identification of the vulnerable input parameter and the configuration of the options that the **DA** tool should use to be able to create the payloads that

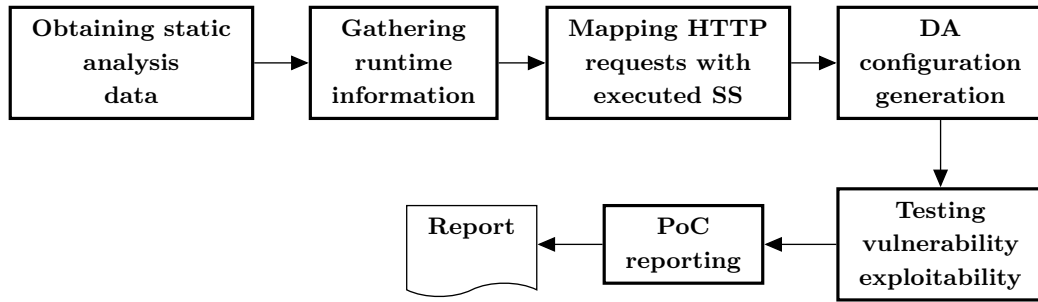


Figure 6.1.: Methodology overview.

effectively and efficiently exploit the target vulnerability (see Figure 6.1).

Obtaining the data needed for the setup of the **DA** tool is a challenging task. For example, the vulnerable web application file stored in the web server may not be directly callable from a simple **URL**, since many times it is only accessible from within a specific context of the main application (as in the case of a WordPress plugin). In order to do this automatically, we have to find a way to learn the process that the main application uses to execute the vulnerable page and be able to repeat it later on with different payloads.

SAST tools typically identify the output vulnerable source code variables, but for the **DA** tools, we need the actual input **HTTP** parameters and their specific values that will be transferred to the source code input variables able to trigger the execution path that reaches the **SS** where the vulnerability is located. To help in this process, the **SAST** tools need to be configured in order to provide not only the information about the **SS**, but also the input variable that is the entry point to exploit the vulnerability.

To address the problems above, we propose a methodology for vulnerability detection, which has six stages, as represented in Fig. 6.1 and discussed in the following subsections:

- 1) **Obtaining static analysis data.** The first stage consists of collecting the web application vulnerability detection data using **SA**. The **SA** can be done manually, which is neither feasible for most projects nor scalable, so this is usually done using a tool (or a set of tools to improve performance), like what developers and security practitioners use so often.
- 2) **Gathering runtime information.** This stage aims at collecting relevant information about the execution of the application. The application can be interacted manually or using an automated tool, like the Acunetix Web Vulnerability Scanner built-in crawler, while executing all the relevant actions (e.g., view, insert and delete data). Either way, the entire interaction is recorded both from within the web server runtime execution trace files (e.g. using the XDebug tool, if the target is a PHP application) and from the **HTTP** interaction (e.g. using a web proxy, like Paros Proxy or Web Scarab).
- 3) **Mapping HTTP requests with trace files.** Since the data in the trace files and in the **HTTP** interactions were obtained independently, using different tools, they are not synchronized. The objective of this stage is to automatically map the **HTTP** requests with the respective web server files that belong to the same user interaction provided by the trace files. The end result are the pairs of **HTTP** requests and the web files they executed. This allows us to know how to navigate from the entry point of the web application to the

vulnerable page. Since the **HTTP** request has the name and value of the input variables, and the trace file has the name of the web server file executed, this mapping also allows us to obtain, for each vulnerable file, the parameters and their concrete values, enabling the execution of the code that triggers the vulnerability.

- 4) **Generating the DA configuration.** This stage aims to automatically generate the configuration of the **DA** tool in order to exploit the vulnerabilities. From the previous stages, we obtained the **URL** (that can be from the main application or from the vulnerable file), the input **HTTP** parameters and their values, and the identification of the vulnerable **HTTP** parameters. From the **SA** output, we can parse the **LOC** of the **SS** in order to find useful attributes, like where each vulnerability is located within the source code file (e.g. a **XSS** inside an **HTML** tag should be treated differently than if it is within a **JavaScript** function) and the type of vulnerability (e.g., a **SQLi SELECT** statement should be exploited differently than an **INSERT** statement). With this knowledge, specific configuration parameters of the **DA** tool can be automatically tweaked in order to improve the likelihood of success and the speed of operation of the **DA** tool.
- 5) **Testing vulnerability exploitability.** This stage checks if each vulnerability is indeed exploitable. The **DA**, with the configurations generated in the previous stage, is executed. If it succeeds in exploiting the vulnerability, the **DA** tool outputs a **PoC**¹. This is a payload that can be manually executed and it is the proof that the vulnerability exists and can be exploited.
- 6) **PoC Reporting.** The information about the vulnerabilities obtained from the previous stages is put together, including data from the **SA** and **DA** tools, in order to have a consolidated report. This report presents a list of all the vulnerabilities found that are proven to be exploitable and their respective **PoC**.

6.1.1. Obtaining Static Analysis Data

To maximize the quality of the output of the **SA**, we can use the results from a set of diverse **SAST** tools (Figure 6.2), instead of using only a single one. In this case, the **SAST** tools perform the analysis of the source code and the results are combined in a way that the same vulnerability detected by several tools is reported only once, as described in Section 4.2. Besides the high **FP** problem, the **SAST** tools tend to miss some vulnerabilities and different **SAST** tools report distinct sets of vulnerabilities, with some overlap [51]. Therefore, combining multiple **SAST** tools has the potential benefit of improving the number of vulnerabilities detected when compared with using a single **SAST** tool due to the complementary nature of the results. However, this process is also likely to generate more **FPS**, since it is well-known that using more tools increase both **TP** and **FP** [205] [119]. To achieve the right balance of **Tps** and **Fps**, in the case study in Section 5.1 we tested different combinations of tools, and were able to provide the best solution depending on the target scenario. This flexibility, allied to our methodology that allows confirming precisely which results are **Tps**, is a step towards the best of both **SA** and **DA** worlds.

¹The **PoC** is an attack against an application that is performed only to prove that it can be done, showing how a hacker can take advantage of a vulnerability in the application.

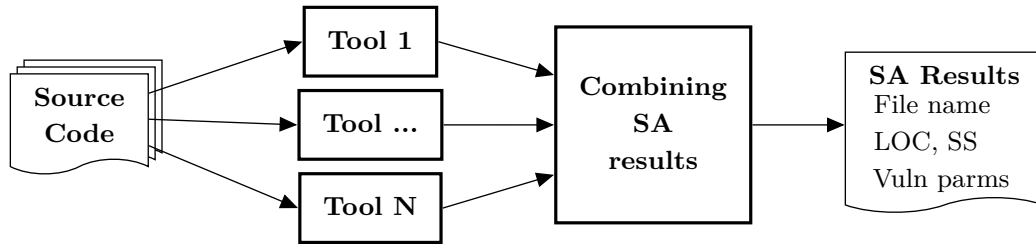


Figure 6.2.: Static Analysis Data.

6.1.2. Gathering Runtime Information

Dynamic analysis has several limitations related with the difficulty in generating test cases (i.e., the inputs, such as [URLs](#) and parameters) that allow the exploitation of a vulnerability. In fact, the coverage and effectiveness of [DA](#) is critically affected by the quality of the test cases used. Our approach to generate good test cases, able to cover nearly 100% of the vulnerable code, is based on real values provided by both the user (or an automated crawler) working with the application (e.g., entering inputs into form fields) and the application itself (e.g., hidden fields and cookies) while it is being executed. To obtain these data, we need to run and interact with the application, while storing the interaction of inputs and respective outputs. This way, we can use the stored interaction data and replay it, while changing some specific values, in order to generate a set of test cases able to attack the potential vulnerabilities detected by the [SA](#).

The methodology for gathering this runtime information consists of following two steps, represented in [Figure 6.3](#).

- 1) **Crawling the application.** The first step is to gather the runtime information of the target application. Before, we need to insert a *proxy* between the user (e.g., the web application developer while testing his work) web browser and the network so that it can capture the [HTTP](#) requests of the interaction. We must also store the server side data. This can be done by means of a web server debug feature, like the [XDebug](#) tool for the PHP Stack Trace. For each [HTTP](#) request, it creates one trace file with relevant data (e.g., [SSs](#) executed and respective parameters). Next, we need a way to interact with the application, which can be done automatically (using an automated crawler) or manually. By crawling the application we mean executing its functions, filling in the form fields and clicking on the hyperlinks in order to test the target module as thoroughly as possible (including the vulnerable parts). While the web application is being interacted, the [HTTP](#) requests and the execution traces are collected and stored in log files independently. To make it easier to synchronize both capture processes, we should crawl the application sequentially instead of using multiple threads.
- 2) **Identifying vulnerable [SSs](#) executed.** The crawling stops only when all the target [SSs](#) identified by the [SA](#) are reached. This is done automatically, since we already have all the necessary information about the [SSs](#) from the [SA](#) and the code executed by the runtime information gathered in the previous step. This process assures the ability to obtain, as close as possible, a perfect coverage of all the vulnerabilities reported by the [SA](#).

The Proxy stores the [HTTP](#) requests in a log file or database. It includes the [URL](#), the request

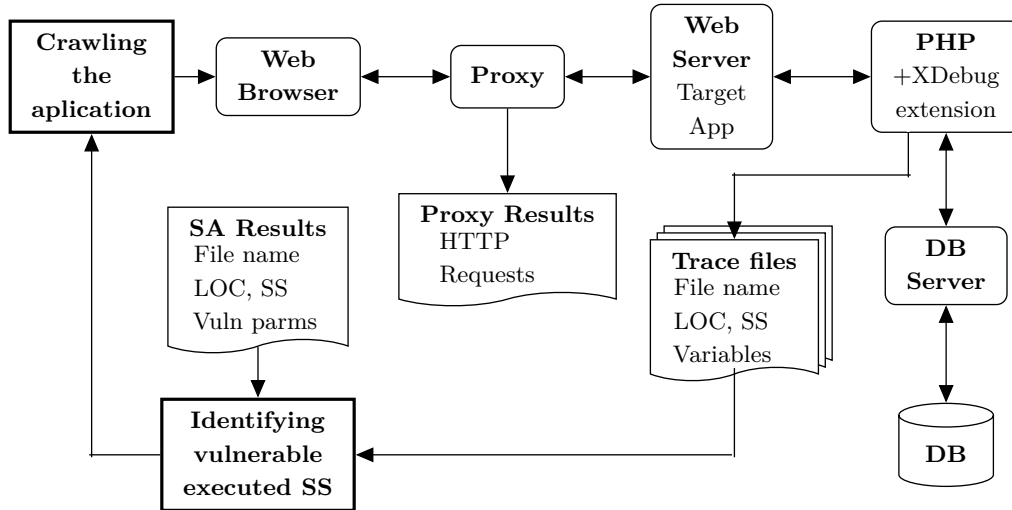


Figure 6.3.: Gathering runtime information.

parameters and values and the timestamp.

The PHP XDebug for each [HTTP](#) request generates a log file with lots of data. The tool can be configured for generating the log file with dynamic name. It can includes the timestamp, the [HTTP](#) request and parameter names and values.

6.1.3. Mapping [HTTP](#) Requests with Trace Files

Since the [HTTP](#) requests and the trace files were obtained using independent software, they are not readily linked to each other. To construct the configuration of the [DA](#) tool, we need to know the [URL](#) that contributed to the execution of the vulnerable server file, the [HTTP](#) input parameters and their values and the identification of the vulnerable input parameter. We also need details about the source code of the vulnerabilities, but they can be obtained by analyzing the [LOCs](#) reported by the [SA](#). The [URL](#) obtained from the [HTTP](#) requests must be mapped with the call to the respective web server file present in the trace file. The parameters and their values are obtained from the [HTTP](#) requests. The identification of the [HTTP](#) vulnerable parameter comes from the information provided by the [SA](#). Since both the trace files and the [HTTP](#) requests have the timestamp information, the trace file originated from the [HTTP](#) request is the first one that was created after the timestamp of that [HTTP](#) request. This is why we should crawl the application sequentially, which should not be a problem, since this process is typically done in the computers of the developers of the web applications.

6.1.4. Generating the [DA](#) Configuration

When entering this stage, we already have the mapping between the report of the vulnerable files and variables from the [SAST](#) tool with the trace files and the mapping of those with the [HTTP](#) requests. Based on these data we can generate a specific configuration of the [DA](#) tool that is able to attack the vulnerability. Depending on the type of vulnerability and on the injection points,

additional parameters may also be used to tailor the payloads to the specific characteristics of the vulnerability and to reduce the number of test cases needed (e.g., level, risk, technique, prefix, suffix, in the case of SQLMap). An example is shown in Listing 6.1, which was generated based on the vulnerability reported by the SA in the WordPress plugin quartz.1.01.1, located at the line 18 of the file all_quotes.php, as illustrated in Listing 6.2.

```
sqlmap.py --threads=10 --dbms=MySQL --batch
-u "http://localhost:81//quartz.1.01.1/wp-admin/edit.php?page=quartz/all_quotes.php&action=delete&paged=1&quote=2"
-p "quote" --level 2 --risk 1 --prefix="" --suffix="--
abc" --technique=BT --cookie="..."
```

Listing 6.1: Example of SQLMap configuration.

From the code of line 18 we extracted that the SS is `$wpdb->query`, the vulnerability is located in the WHERE clause of a SQL DELETE statement and the vulnerable variable is `$_REQUEST['quote']` enclosed in single quotes (`'`). From this we generate the following part of the DA configuration “`--prefix="" --suffix="-- abc`” for creating and injection point (`WHERE ID='2' <injection_point> -- abc'`) between the single quotes of the SQL statement. The option “`--technique=BT`” indicates to the DA to perform *boolean-based blind injection* (B) and *time-based blind injection* (T). We selected these type of injection because we are exploiting a DELETE SQL statement and the results the execution of this statement may not be visible in the HTTP responses of the server. These options takes more time in the DA but guaranties success when exploiting of the vulnerabilities that are in fact exploitable.

```
17 if($_REQUEST['action'] == 'delete') {
18     $wpdb->query("DELETE FROM {$wpdb->prefix}quartz_quote WHERE ID='$_REQUEST['quote']");
```

Listing 6.2: Slice of code of the file all_quotes.php and plugins quartz.1.01.1.

Listing 6.3 shows data collected when the SS was executed during the crawling process. The item HTTP request contains the parameters and values. It include the four parameter: page (quartz/all_quotes.php), action (delete), paged (1) and the quote (2), the vulnerable parameter. The item SS contains the name of the SS executed and the SQL statements executed. From this we generate the following part of the DA configuration “`-u "http://localhost:81//quartz.1.01.1/wp-admin/edit.php?page=quartz/all_quotes.php&action=delete&paged=1"e=2"`” and “`-p "quote"`”.

```
HTTP: localhost:81//quartz.1.01.1/wp-admin/edit.php?page=quartz/all_quotes.php&action=delete&paged=1&quote=2
SS: wpdb->query; ["$query = 'DELETE FROM wp_quartz_quote WHERE ID=\\'2\\''"]
```

Listing 6.3: Data collected during the crawling of the plugin quartz.1.01.1.

The remaining parameters and values in the configuration: “`--threads=10`”, “`--dbms=MySQL`” “`--batch`”, “`--level 2`” and “`--risk 1`” are used to tune the DA and their meaning are further explained in Section 6.2.4.

6.1.5. Testing Vulnerability Exploitability

In this stage, the **DA** is executed, using the configuration created in the previous stage, in order to confirm (in case of successful exploitation) the vulnerabilities found by the **SAST** tools. Even if the **DA** cannot exploit the vulnerability, there is no guarantee that it is unexploitable (nor a **FP** of the **SA**). If these failed cases are very important for the developer, they must be verified manually. If a vulnerability is found by the **SAST** tool and the **DA** can exploit it, then it is proven to be exploitable. On success, the **DA** tool provides a **PoC** on how the vulnerability can be exploited. This is usually a **URL** link that can be used in a web browser or a block of code that can be executed autonomously in a terminal window. This **PoC** is the proof that the vulnerability can indeed be exploited. A **PoC** example with a **SQLMap** usage is shown in Listing 6.4.

```
SQLMap output:
GET parameter 'quote' is vulnerable
Execution time: 117s; Total of HTTP(s) requests: 46
Parameter: quote (GET), Type: AND/OR time-based blind
Payload: page=quartz/all_quotes.php&action=delete&paged=1 &quote=2' AND(SELECT * FROM(SELECT(SLEEP(5)))glYA)-a
Vector: AND (SELECT * FROM (SELECT(SLEEP([SLEEPTIME]-(IF([INFERENCE],0,[SLEEPTIME]))))) [RANDSTR])
```

Listing 6.4: Example of **PoC** report for the quartz.1.01.1 plugin.

6.1.6. PoC Reporting

Reporting results involves communicating the findings of the vulnerability detection in a manner that makes sense to the target audience, like system owners and developers. The report should include information about both the vulnerabilities successfully exploited and the vulnerabilities that were not exploited by the **DA**. For the vulnerabilities successfully exploited, it includes the number of requests, the execution time, the attack vectors (i.e., types of attack) and the payloads (i.e., data needed to replay the attacks). For the vulnerabilities detected by the **SA** but that could not be exploited by the **DA**, it only includes the number of requests and the execution time. It is important to emphasize that we do not consider these cases as **FP** until there is a manual confirmation by an expert. Regarding this situation, a real benefit of using our methodology is that this number is greatly reduced when comparing with typical procedures, as we will see in Section 6.3.1.

6.2. Instantiation and Experimental Setup

This section presents a concrete instance of our generic approach for the PHP language, **SQLi** vulnerabilities and WordPress plugins but the concepts behind it are generic and are applicable to other web languages. It also provides concrete examples on how the different steps of the approach were executed in practice.

Table 6.1.: List of WordPress Plugins with first-order [SQLi](#).

Plugin	TP	FP	SS	Files	LOC
ajaxgallery.3.0	2	0	5	4	388
another-wordpress-classifieds-plugin.2.2.1	2	0	107	79	25,767
calculated-fields-form.1.0.10	2	0	24	7	1,993
collision-testimonials.3.0	9	1	65	4	1,562
community-events.1.2.9	3	0	59	9	4,588
contact-form.2.7.5	4	0	14	10	1,464
contus-hd-flv-player.1.3	9	2	58	11	3,119
contus-video-gallery.2.8	3	2	128	79	14,544
couponer.1.2	2	0	13	5	572
cp-reservation-calendar.1.1.6	4	0	18	6	1,734
dukapress.2.5.9	1	0	31	25	9,346
dynamic-font-replacement-4wp.1.3	4	0	6	9	738
easy-career-openings	5	0	9	8	619
evarisk.5.1.3.6	14	0	655	237	195,147
eventify.1.7.f	5	0	21	7	4,261
events-registration.5.44	52	0	84	40	11,626
flash-album-gallery.2.55	1	9	107	81	16,748
forum-server.1.7.1	8	4	104	10	4,407
fs-real-estate-plugin.2.06.01	52	0	290	31	4,353
gallery-images.1.0.1	14	0	114	10	6,352
global-content-blocks.1.2	3	0	18	8	1,005
ip-blacklist-cloud.3.4	7	5	125	17	8,019
ip-logger.3.0	3	0	89	75	6,088
js-appointment.1.5	85	0	113	42	18,272
knr-author-list-widget.2.0.0	1	0	4	3	1,124
levelfourstorefront.8.1.14	31	57	467	183	49,502
media-library-categories.1.0.6	3	0	5	8	1,554
mystat.2.6	3	0	13	38	4,051
mz-jajak.2.1	8	0	30	4	988
newsletter.3.6.4	1	0	62	102	12,133
odihost-newsletter-plugin	17	1	50	24	3,971
oqey-headers.0.3	0	1	4	5	423
pie-register.2.0.18	4	2	25	32	15,837
profiles.2.0.RC1	10	0	14	14	1,014
quartz.1.01.1	1	0	9	5	599
scormcloud.1.0.6.6	4	0	27	34	5,810
sendit.2.1.0	14	0	60	14	3,122
sermon-browser.0.43	1	35	179	19	14,788
sh-slideshow.3.1.4	7	0	15	9	2,729
slider-image.2.6.8	1	10	144	8	5,062
stripshow.2.5.2	3	0	17	56	9,241
wp125.1.5.3	0	1	42	6	859
wp-championship.5.8	14	3	206	24	7,248
wp-ds-faq.1.3.2	8	0	23	2	1,197
wpforum.1.7.8	12	4	115	10	4,151
wp-menu-creator.1.1.7	11	0	43	15	2,393
wp-powerplaygallery.3.3	1	0	28	14	2,757
wp-predict.1.0	8	0	44	1	10,120
wp-symposium.14.10	6	2	903	117	49,119
Total (49)	462	140	4,786	1,561	552,504

SS - Sensitive Sink.

6.2.1. Obtaining Static Analysis Data

To demonstrate our approach, we used the dataset created in Section 4.2, built using five SAST tools that were individually run to find SQLi and XSS vulnerabilities in 134 WordPress plugins. The outputs of the tools were combined and annotated, so each candidate vulnerability was manually reviewed to identify if it was a TP or a FP. This dataset is perfect to evaluate our blended methodology because with it we have a corpus with many vulnerabilities that have already been manually confirmed by experts.

Second-order SQLi vulnerabilities are a serious threat to web applications and are more difficult to detect than first-order SQLi vulnerabilities [206]. The SAST tools used for creating the dataset we used have serious limitation detection second-order SQLi vulnerabilities. For example, most tools report a second-order SQLi vulnerability whenever data coming from the database is directly used for creating SQL statements without confirming that these data was stored without validation. For this reason, we focused only on the first-order subset of SQLi vulnerabilities in the dataset. The detection results of the SAST tools for these vulnerabilities are summarized in Table 6.1, which includes the total number of potential SS in the plugins that are about 8 times more than the total number of SA results (TPs+FPs). Therefore, the benefit of having SA to help “focus” the DA is enormous.

When looking for SQLi vulnerabilities in these plugins, the SAST tools generated 602 alarms. The question now is, how to automatically separate the real vulnerabilities from the false alarms, since we know from the manual annotation that only 462 are TPs that can be exploited and 140 are FPs. Similar to real world scenarios, we do not have such annotations so, our goal here is to demonstrate that the process of confirming the exploitability of the vast majority of the cases can be effectively done using our approach, leaving just a few vulnerabilities that were not exploited (thus fostering trust in the approach).

6.2.2. Gathering Runtime Information

To gather the runtime information of the WordPress plugins shown in Table 6.1, we need to install WordPress, install the plugins, setup the testing environment and then run the plugins from within the WordPress environment:

- 1) **Installing the WordPress and plugins.** First we need to install and setup the WordPress application, and then install the plugins. To avoid contamination from the presence of other plugins and the bias they may inflict to the database during the execution of the tests, we install each plugin in its own instance of the WordPress.
- 2) **Setting up the debugger.** XDebug is an extension for PHP to assist with debugging and development [207]. Therefore, for gathering the execution trace files of the plugins, we activated the XDebug extension and configured it to create one debug file per HTTP request.
- 3) **Setting up the proxy.** For web proxy we have chosen the OWASP ZAP [208], which is one of the most popular free security tools. When it is used as a web proxy, it is able to save the HTTP user interaction (requests and responses) in its database to be queried later.

- 4) **Crawling the applications.** Since commercial automated crawlers are quite expensive, we have chosen to crawl the plugins manually. This way, this task takes a fair amount of time, especially when dealing with plugins where the target features are only triggered automatically (e.g., when a timer expires) or externally (e.g., when an user clicks on an email link to continue an action initiated in the plugin to reset a password) and not from the user interface of the plugin or the WordPress.
- 5) **Identifying the execution of the vulnerable `SSs`.** `XDebug` trace files include lots of data about the execution of the web application. As we do not need all these data, we used the `XDebug` Trace Manipulator (`Xtm` tool [209]) to parse, filter and format the trace file data and create a new file with only the data that we need: the file name, `LOC`, variables and values for the executed database functions (i.e., `SQLi` related `SSs` function calls, like `mysql_query`). Since the results of the `SAST` tools also include the vulnerable file name, `LOC`, `SSs` function and variables, identifying the executed `SSs` is done through a sequential search of the data reported by the `SAST` tools in the `Xtm` file generated by `XDebug`. To fully automate this process, we created a `Python` script that gathers all the trace files and automatically performs this process for all of them at once.

It is important to mention that, a manual crawling process requires an user with the ability to explore all elements (e.g., links, buttons and input fields) in the plugins. The user has to be familiar with WordPress and a good knowledge about the functionalities of the plugins. In fact, different plugins add menu options in different parts of the WordPress environment. Therefore, the user has to be aware of this. With the plugin installed and configured, the user starts interacting from the login form by submitting the admin credentials. WordPress starts the administration dashboard. From this point, all visible buttons related with the plugin (i.e., excluding general WordPress options) have to be clicked, and all the forms with input elements must be filled and submitted. Note, that for some plugins, we have to create one or more WordPress/plugin users with different roles (e.g., Customer and Shop Manager for an e-commerce plugin) to make visible and allow access to some parts of the plugin. In these cases, the user crawling the plugin has to log in as these users and navigating the plugin, executing every feature.

The manual crawling was done by an application user with full privileges, performing all the user interface options, such as view, insert, delete, alter, sort, filter, and search data. With our approach, we are able to obtain the data about the vulnerabilities triggered during the crawling (see step 5). Therefore, the crawling should continue until 100% coverage of the vulnerabilities is reached. However, in our experiments, the manual crawling did not reach 100% for some plugins (see tables 6.4 and 6.6), many times due to a specific execution state (e.g., the presence of specific records in the database, cookie values, etc.). For instance, the actions performed when deleting an item may vary according to the data stored in the back-end database (e.g., the presence or absence of a dependent record). Moreover, there are links and inputs that only exist for concrete states of the data in the database. For example, delete and update operation links exists if there are data in the database or if the data is in a specific state (e.g., in an e-commerce web application, orders in the state “in process” can no longer be edited by the client). To be able to reach all the pieces of code of all those actions, it may be necessary to create several specific examples in the database, which requires a deep understanding of the inner workings of the plugin, which is usually the case when the plugin developer is testing it, before deployment.

Since the crawling may damage the integrity of the database data, we automatically made, with a shell script, a backup at the beginning and restored it before each execution of the crawling process. The time taken for the crawling step varies a lot depending on the complexity of the plugins. During our tests, we spend between one minute for the simpler plugins, where we only had to input one or two values, to about 15 minutes for the most complex plugins, where we had to navigate through many pages and fill in several input values: choosing options available from menus, buttons, and hyperlinks, data entry on forms, checking emails to confirm operations, drag and drop data items up and down. In other situations, due to the dynamic interface of some of the plugins, we had to insert specific data in the database in order to make some user options available.

6.2.3. Mapping HTTP Requests with Trace Files

During this stage, we mapped the trace files with the [HTTP](#) requests gathered by the proxy, using the timestamp information they both have. The matching trace file is the first one created after the [HTTP](#) request, since we are executing all the actions sequentially, without threads. Listing 6.5 shows several trace files names generated during the crawling process for the `gallery-images.1.0.1` WordPress plugin. The trace files names have the following template name: “name_<timestamp(integer part)>_<timestamp(decimal part)>_<HTTP request>”. We removed from the [HTTP](#) request the prefix “localhost:81/” and replaced invalid characters for file names (e.g., “/”, “&”, “?”, “:”) by an underscore “_” character. The decimal part of the timestamp has 6 digits of granularity (microseconds). We included the [HTTP](#) request in the trace file names for cross validation when mapping the trace file names with [HTTP](#) requests.

```
1 ...
2 trace_1555064612_461462__wordpress44_gallery-images_1_0_1_wp-admin_admin_php_page=gallerys_huge_it_gallery_id=4
   _task=apply.xt
3 trace_1555064663_540711__wordpress44_gallery-images_1_0_1_wp-admin_admin_php_page=gallerys_huge_it_gallery_id=4
   _task=apply.xt
4 trace_1555064689_323058__wordpress44_gallery-images_1_0_1_wp-admin_admin_php_page=gallerys_huge_it_gallery_task=
   edit_cat_id=4_remove-slide=26.xt
5 trace_1556466067_750809__wordpress44_gallery-images_1_0_1_wp-admin_admin_php_page=gallerys_huge_it_gallery_id=5
   _task=apply.xt
6 trace_1556466084_164492__wordpress44_gallery-images_1_0_1_wp-admin_admin_php_page=gallerys_huge_it_gallery_id=5
   _task=apply.xt
7 trace_1556466094_063106__wordpress44_gallery-images_1_0_1_wp-admin_admin_php_page=gallerys_huge_it_gallery_id=5
   _task=apply.xt
8 ...
```

Listing 6.5: Partial list of trace files for the plugin `gallery-images.1.0.1`.

The [HTTP](#) request in Listing 6.6 was mapped with the trace file name in line 4 of Listing 6.5, as this is the trace file with the lowest timestamp “1555064689_307” (3 digits of granularity, milliseconds) that is greater than the timestamp of the [HTTP](#) request “1555064689_323058”. In this case, the trace file was created just over 16 milliseconds ($323058 - 307 \times 1000$) after the [HTTP](#) request. For cross validation, we verify in the content of the trace file mapped if the target [SS](#) was in fact executed.

```
Timestamp...: 1555064689_307
HTTP request: localhost:81/wordpress44_gallery-images.1.0.1/wp-admin/admin.php?page=gallery_huge_it_gallery&task=
edit_cat&id=4&removeslide=26
```

Listing 6.6: Example of [HTTP](#) request log.

6.2.4. Generating the DA Configuration

To test a vulnerable parameter, we can use automated tools, such as [SQLMap](#), [bsqlbf-v2](#) or [darkjumperv5.7](#). Our choice was [SQLMap](#), because it is a free, fully-featured, configurable and widely used tool. In order to configure its internal behavior, we need to provide information about the target application: `-u` specifies the [URL](#), `-data` the [HTTP](#) POST request string, and several other options, such as the `-cookie` when we need authentication. The exact configuration must be tailored according to the specificity of the plugin and the vulnerability.

6.2.4.1. Target Vulnerability Execution

Depending on the WordPress plugin, we have two ways to execute its code: 1) direct call and 2) indirect call. A plugin may use any one of these ways or even both of them (one for some parts of the code and the other for other parts of the code).

Direct call is the easier to execute. The [URL](#) is composed by the *main address* and the [GET](#) parameters separated by the “?” symbol. The *main address* is composed by the base [URL](#) of the application (e.g., [www.mysite.com](#)) concatenated with the plugins folder (`wp-content/plugins`) and the full path name of the file of the plugin containing the vulnerability (e.g., `ajaxgallery/utils/deleteItem.php`). The first [GET](#) parameter is indicated by `?`, while the subsequent parameters are indicated by `&`. The [GET](#) parameters take the format “`?par1=value1&par2=value2&par3=value3`” and are used to send data to the web application. The code in [Listing 6.7](#) shows an example.

```
http://mysite/wp-content/plugins/ajaxgallery/utils/deleteItem.php?itemId=3
```

Listing 6.7: [URL](#) of direct call to execute part of a plugin code (deleting image).

In the example, the [URL](#) contains a [GET](#) parameter (`itemId=3`). The [URL](#) is the entry point of the plugin and when executed it deletes the images of the gallery identified by the `itemId` with the value 3. To be executable, the plugin may require other parameters and variables with concrete values to represent the possible execution paths containing the [SS](#). For some cases, these data can be determined statically and for other cases, it requires our crawling process to collect these data.

The **Indirect call** is needed when the plugin has to be called from within the WordPress environment. To obtain the information necessary to execute this case, we also have to use the [HTTP](#) request obtained from the database of the web proxy. The construction of the [URL](#) is

more complex in this case, since the [URL](#) is virtual and we need to provide the correct WordPress environment variables that will trigger the execution of the target vulnerable file with the right parameters. It has two variations:

- The first consists of providing a specific WordPress page, like the admin page (e.g., `wp-admin/admin.php?page=<name>`, where `<name>` is the name of the page to execute).
- The second corresponds to an [URL](#) that is dynamically generated by the WordPress interface. For example, hyperlinks to edit and delete items displayed in the web browser. Listing 6.8 shows such an example where the event identified by the `deleteevent=2` of the `community-events-events` page is deleted.

```
http://mysite/community-events.1.2.9/wp-admin/admin.php?page=community-events-events&deleteevent=2
```

Listing 6.8: [URL](#) of indirect call to execute part of a plugin code (deleting event).

6.2.4.2. SQLMap Internal Options

The options about the internal behavior of **SQLMap** are organized in several categories, including: general, injection, detection, techniques and optimization. From the general options, we used the `-batch` to never ask for user input, which allows automating the process (see Section 6.2.5 for details). Next, we list the **SQLMap** options that we used the most in the **DA** part.

Injection: Specifies which parameters to test for, the type of back-end database, and defines custom injection payloads:

- `dbms=VALUE`. Forces back-end [DBMS](#) to this value. Since the WordPress uses **MySQL**, that was our choice.
- `p`: testable parameter(s). In the execution path of a vulnerable [SQL](#) statement there may exist several vulnerable parameters of other [SQL](#) statements. However, in our experiments we tested one parameter at a time.
- Injection payload `prefix` and `suffix` strings. These options are very useful when the user knows the syntax of the [SQL](#) statement [210], like in our work from the **SA**. Providing values for the prefix and suffix prevents the use of brute force trying all the combinations. By analyzing the syntax of the vulnerable [SQL](#) statement we are able to automatically provide values for the prefix and suffix options.

To clarify the importance of the prefix and suffix, we present in Listing 6.9 some common situations taken from the plugins. All statements have an injection point in the **WHERE** clause. However, the predicate in line 1 uses single quotes “'” delimiting the PHP variable `$id`, in line 2 it has parenthesis and in line 3 it has nothing. For example, considering the first [SQL](#) statement, by providing a single quote “'” for the prefix, the string “`-- abc`” for the suffix and the value `1` for the parameter `id`, the injection pattern of the **WHERE** clause becomes “`WHERE id='1' <PAYLOAD>-- abc'`”. The suffix string “`-- abc`” is used to avoid eventual errors by commenting the rest of the [SQL](#) statement. In the second situation, we should use the parenthesis “)” and, for the third situation, the null string “” for both for the prefix and the suffix.

```

1 DELETE FROM $wpdb->events WHERE id='$id'
2 SELECT * FROM events WHERE id=($id) ORDER BY $field $op
3 UPDATE events SET name = '$event_name', $field = $field_value WHERE id=$id

```

Listing 6.9: SQL statements with injection points.

Detection: Options to customize the detection phase:

- **risk.** The likelihood of a payload to damage the data integrity (1-3, default 1). Risk value 1 is innocuous for the majority of SQL injection points. Risk value 2 adds to the default level the tests for heavy query time-based SQL injections, and value 3 adds also OR-based SQL injection tests (High risk). We started with 1 and increased the value if nothing was detected. The reason this is in the highest risk level is because injecting OR payloads in certain queries can actually lead to update or delete all entries in database tables. Changing data in the database is never what we would want unless we are testing a throw-away environment and database. If we were to do that in a production environment, it could have disastrous consequences.
- **level.** Level of tests to perform (1-5, default 1 < 100 requests). This option limits the maximum number of test cases tried to exploit the vulnerable parameter. We start with the default value of 1 and increase the value by one whenever the vulnerability is not successfully exploited.

6.2.5. Testing the Vulnerability Exploitability

To check if a vulnerability is exploitable, we executed `SQLMap` with the configuration generated in the previous stage (Section 6.2.4). The creation of the `SQLMap` configuration, including the definition of the `URL` (with direct or indirect call) and the selection of the options (injection, detection, techniques and optimization) is done automatically by a set of `Python` scripts.

For each vulnerability, `SQLMap` may provide two responses:

- 1) **The tested parameter is vulnerable.** In this case, the output is saved in a text file.
- 2) **All tested parameters do not appear to be injectable.** In this case, we change one `SQLMap` option at a time, restore the database and re-execute `SQLMap`. When all test options were tried without success, we consider that the vulnerability is not exploitable, and we check it manually to confirm. The details on how the options are mutated are the following:
 - The **risk** detection option is the first one to be changed. We start by increasing it to **risk=2** to test with time based techniques and then to **risk=3** to test with OR-based techniques.
 - The next option to be changed is the **level** that has 5 as maximum.
 - Afterwards, we remove the prefix and suffix configurations, which tests all prefix and suffix combinations.
 - Finally, we try to test with all injection techniques.

Listing 6.10 shows an example of one of our `SQLMap` tests (items “`SQLMap usage`” and “`SQLMap`”

output”) including the output. We can see that SQLMap executed 46 [HTTP](#) requests (i.e., test cases) to confirm that the tested parameter `quote` is vulnerable.

6.2.6. PoC Reporting

After executing all previous stages for each vulnerability reported by the [SA](#), a comprehensive report fusing the data gathered from all the stages is generated (see [Listing 6.10](#) for an example). The item “SA” shows data from the [SA](#), including the vulnerable file, number of LOC, sensitive sink (`wpdb->query`), vulnerable parameter (`quote`). It also includes the LOCs (not all in this case) where we see that for triggering the [SS](#) is required the parameter `action` with the value `delete`. The item “HTTP”, shows the request triggering the sensitive sink with parameter’s values. The item “SS”, contains the vulnerable code executed during the crawling of the application, with the value 2 for the vulnerable parameter. The item “SQLMap usage”, shows the command used to exploit the vulnerable code. Finally, the last item “SQLMap output” shows part of the results of running the [DA](#). In this case SQLMap was successfully exploited the vulnerable code.

```
SA: all_quotes.php; 18; wpdb->query; quote
    if($_REQUEST['action'] == 'delete') {
        $wpdb->query("DELETE FROM {$wpdb->prefix}quartz_quote WHERE ID='$_REQUEST[quote]'");
HTTP:
    HTTP://localhost:81//quartz.1.01.1/wp-admin/edit.php?page=quartz/all_quotes.php&action=delete&paged=1&quote=2
SS:
    wpdb->query; ["$query = 'DELETE FROM wp_quartz_quote WHERE ID=\\'2\\''"]
SQLMap usage:
    sqlmap.py --threads=10 --dbms=MySQL --batch
    -u "http://localhost:81//quartz.1.01.1/wp-admin/edit.php?page=quartz/all_quotes.php&action=delete&paged=1&quote=2
      " -p "quote" --level 2 --risk 1 --prefix="" --suffix="--
      abc" --technique=BT --cookie="..."
SQLMap output:
    GET parameter 'quote' is vulnerable
    Execution time: 117s; Total of HTTP(s) requests: 46
    Parameter: quote (GET), Type: AND/OR time-based blind
    Payload: page=quartz/all_quotes.php&action=delete&paged=1 &quote=2' AND(SELECT * FROM(SELECT(SLEEP(5)))g1YA)-a
    Vector: AND (SELECT * FROM (SELECT(SLEEP([SLEEPTIME]-(IF([INFERENCE],0,[SLEEPTIME]))))) [RANDSTR])
```

Listing 6.10: Example of a complete [PoC](#) report for the `quartz.1.01.1` plugin.

6.3. Results and Discussion

This section presents and discusses the results of experimental evaluation. As mentioned in [Section 6.2.1](#), for the purpose of this evaluation, we have the vulnerability data annotated so we can identify the correctness of the results reported by the tools and by our approach. To understand the effectiveness of our approach, we also discuss how the results compare with those obtained by directly using the outcome of the [SA](#) (like the [URL](#), vulnerable parameters and parameters determined statically from the path) as the input of the [DA](#) tool.

6.3.1. Overall Results

Table 6.2 shows the overall results of our blended approach. The first column indicates the type of call (direct or indirect) needed to execute the plugin code (as discussed in Section 6.2.4).

Table 6.2.: Overall Results of our Blended Methodology.

Call Type	TPs			FPs			Total			
	SA	C	DA	SA	C	DA	SA	C	DA	%
Direct	80	70	70	18	18	18	98	88	88	89.8
Indirect	382	287	266	122	110	108	504	397	374	74.2
Total	462	287	336	140	110	126	602	397	462	76.7

SA-SA tools results. C - Crawling. DA-SQLMap results % - DA/SA \times 100.

For the TPs, column “SA” has the vulnerabilities correctly detected by the SA stage, column “C” shows those from which we could obtain the configuration parameters during the crawling, and column “DA” presents those that could be exploited by the DA tool, which represent the vulnerabilities that could be confirmed as such by our approach. As for the FPs, column “SA” has the miss classified as vulnerabilities by the SA stage, column “C” shows those from which we could obtain the configuration parameters during the crawling, and column “DA” presents those that could not be exploited by the DA tool, so they are the results that our approach indicates as not being a vulnerability. The “Total” has the compound of the positive results: column “SA”, has all the vulnerabilities identified (both correctly and incorrectly) by the SA stage, column “C” shows the situations from which we could obtain the configuration parameters during the crawling, column “DA” presents those that were correctly identified by our approach (either as a vulnerability or as false alarm), and column “%” has the ratio of the results correctly identified by our approach over the total number of positives given by the SA stage.

In short, by using our methodology, 76.7% (462) of the vulnerabilities reported by the SA could be confirmed either as TPs or FPs, leaving only 140 (602-462) out of 602 vulnerabilities to be checked manually (Table 6.2). A discussion about the reasons why these vulnerabilities were failed to be confirmed by the DA tool is presented in the following.

6.3.2. Testing the Vulnerability Exploitability

Table 6.3 shows the results of the TPs successfully exploited using the direct call, and Table 6.4 those using the indirect call. In these tables, we can see the vulnerabilities found by the SAs and the subset that SQLMap could confirm. The last column shows the total number of vulnerabilities that were confirmed to be exploitable. The columns are also organized by type of SQL statement that could be exploited: SELECT (S), INSERT (I), DELETE (D), UPDATE (U) and M for multiple SQL statements. The multiple SQL statements identify situations where different SQL statements share the same vulnerable variables in the same execution path.

For direct call group (Table 6.3), our crawling process collected all data required to successfully exploit 70 TPs. Table 6.3 also shows the TPs (27, in column B) that could be exploited based on the data collected statically. Therefore, parameters and concrete values could be collected directly

Table 6.3.: TPs Exploited With Direct Call.

Plugin	SA	Our methodology							
		Crawl		<i>S</i>	<i>I</i>	<i>D</i>	<i>U</i>	<i>M</i>	<i>T</i>
		<i>A</i>	<i>B</i>						
ajaxgallery.3.0	2	1	1			1	1		2
another-wordpress-classifieds-plugin.2.2.1	1	1	0	1					1
contact-form.2.7.5	1	0	1				1		1
couponer.1.2	2	2	0	2					2
events-registration.5.44	6	1	5	2	4				6
fs-real-estate-plugin.2.06.01	2	0	2					2	2
global-content-blocks.1.2	1	0	1		1				1
ip-logger.3.0	2	0	1	1					1
levelfourstorefront.8.1.14	31	0	23	10			13		23
media-library-categories.1.0.6	1	0	1	1					1
odihost-newsletter-plugin	6	5	1		4		2		6
profiles.2.0.RC1	9	9	0	2		1	2	4	9
sendit.2.1.0	7	0	6		2		3	1	6
wp-menu-creator.1.1.7	6	5	1					6	6
wp-powerplaygallery.3.3	1	1	0		1				1
wp-symposium.14.10	2	2	0				2		2
Total (16)	80	27	43	19	12	2	24	13	70

S-Select, I-Insert, D-Delete, U-Update, M-Multiple SQL, T-Total.

A - Cases where runtime data could be collected statically based on SA results.

B - Cases where crawling is mandatory for collecting runtime data.

from the SA results or collected by analyzing the code in the possible vulnerable execution paths reported by the SA. For example, from the source code line `if (isset($_POST['save']))` in an possible execution path, we extracted the required parameter `POST save`. For the direct call group, we can see that our approach failed to exploit 10 confirmed vulnerabilities. One is in the `ip-logger.3.0` plugin that requires an account in the `mretzlaff.com` web site. Another eight are in the `levelfourstorefront.8.1.14` plugin that requires a PayPal account. Finally, the last one is on multiple SQL statements in the `sendit.2.1.0` plugin. By analyzing its source code (used to insert new subscribers), we found a conditional statement where the vulnerable code is only executed when the email of the new subscriber does not exist in the database. In order to be able to exploit the vulnerability, the DA would need to use an email that do not exist in the database, which SQLMap cannot obtain automatically.

Regarding the indirect call group (Table 6.2), 95 (382-287) SA results were not covered (i.e., not executed) during the crawling process, so they could not even be analyzed by SQLMap. In fact, the crawling procedure was only able to cover 69.6% of the vulnerable code. As expected, it may be difficult for the crawling to cover all the TPs, given the specific needs of the plugins. For example, some plugins require a commercial license to activate some features (e.g., 42 TPs of the `js-appointment.1.5` plugin are located in its commercial part); the plugins `levelfourstorefront.8.1.14` and `events-registration.5.44` require a PayPal account to make payments; plugins, such as `newsletter.3.6.4`, execute cron jobs to periodically send emails; other plugins, like `wp-championship.5.8`, notify their users according to the state of the database data. These are all interesting problems that need to be further researched in order to improve the effectiveness of the crawling process.

Table 6.4.: TPs Exploited With Indirect Call.

Plugin	SA	Our methodology						
		<i>Crawl</i>	<i>S</i>	<i>I</i>	<i>D</i>	<i>U</i>	<i>M</i>	<i>T</i>
calculated-fields-form.1.0.10	2	2			1	1		2
collision-testimonials.3.0	9	9	2		6		1	9
community-events.1.2.9	3	3					3	3
contact-form.2.7.5	3	3					3	3
contus-hd-flv-player.1.3	9	9	5	3		1		9
contus-video-gallery.2.8	3	3		1		2		3
cp-reservation-calendar.1.1.6	4	4		2	1	1		4
dukapress.2.5.9	1	1			1			1
dynamic-font-replacement-4wp.1.3	4	4	1	1	1	1		4
easy-career-openings	5	5	3	1		1		5
evarisk.5.1.3.6	14	7		3		1	3	7
events-registration.5.44	46	38	15	1	5	7	1	29
forum-server.1.7.1	8	8			3	5		8
fs-real-estate-plugin.2.06.01	50	40	1	14	2	9	14	40
gallery-images.1.0.1	14	11		1	1	9		11
global-content-blocks.1.2	2	2		1		1		2
ip-blacklist-cloud.3.4	7	4		1			2	3
js-appointment.1.5	85	31	20	3	4	4		31
knr-author-list-widget.2.0.0	1	1				1		1
media-library-categories.1.0.6	2	2				2		2
mz-jajak.2.1	8	8		5	1	2		8
odihost-newsletter-plugin	11	11	1	5	2	2	1	11
pie-register.2.0.18	4	4		1	1	2		4
profiles.2.0.RC1	1	1					1	1
quartz.1.01.1	1	1					1	1
scormcloud.1.0.6.6	4	4				1		1
sendit.2.1.0	7	7		2	2	3		7
sermon-browser.0.43	1	1					1	1
sh-slideshow.3.1.4	7	7		1	1	1	4	7
slider-image.2.6.8	1	1			1			1
stripshow.2.5.2	3	3		1	1	1		3
wp-championship.5.8	14	14		4	4	4	1	13
wp-ds-faq.1.3.2	8	8			4	2	2	8
wpforum.1.7.8	12	7		2	2	2	1	7
wp-menu-creator.1.1.7	5	5					5	5
wp-predict.1.0	8	7		2			5	7
wp-symposium.14.10	6	4					4	4
Other (3)*	9	3						0
Total (40)	382	287	48	55	44	66	53	266

SA-TPs of the SA. S-Select, I-Insert, D-Delete, U-Update, M-Multiple SQL, T-Total.

*Plugins that could neither be exploited with our methodology nor manually.

Considering the 287 **TPs** crawled by our approach, **SQLMap** was unable to exploit 21 (287-266) (Table 6.4), but there are several reasons for this. For instance, **SQLMap** was not able to exploit nine vulnerabilities of the `events-registration.5.44` plugin because it has a **CAPTCHA** challenge. One missed vulnerability was found in the `newsletter.3.6.4` plugin. It occurs in an **UPDATE** statement which aims at resetting the values of one record of the table, but by injecting the payload “OR 1=1” in the `id` parameter, all records of the table were updated. In general, it is difficult for **DA** tools to detect **SQL** commands that change the database, like **INSERT**, **DELETE** and **UPDATE**, because of the limited feedback they provide and the changes they introduce for the following tests. The three vulnerabilities unexploited of the `eventify.1.7.f` plugin were due to the use of **nonces** to avoid the replay of the same operation. Finally, when exploiting one vulnerability in the `ip-blacklist-cloud.3.4`, the database data is deleted and the website crashes, preventing the detection.

6.3.3. Testing the Non-Exploitability of FPs

In the direct call group, our approach was able to confirm all the **FPs** as such (Table 6.5), given that **SQLMap** was unable to exploit them. For the indirect call group (Table 6.6), it was also able to confirm all, except 14 situations that were excluded earlier in the crawling process due to a **CAPTCHA** challenge.

Table 6.5.: FPs Tested With Direct Call.

Plugin	SA	Our methodology					
		<i>S</i>	<i>I</i>	<i>D</i>	<i>U</i>	<i>M</i>	<i>T</i>
levelfourstorefront.8.1.14	17	14	2		1		17
oqey-headers.0.3	1				1		1
Total (2)	18	14	2		2		18

S-Select, I-Insert, D-Delete, U-Update, M-Multiple SQL, T-Total.

Based on the annotated dataset (i.e., we knew the **FPs**), we expected **SQLMap** to fail in exploiting some of the vulnerabilities reported by the **SA**, such as the **FPs**. However, in a real world situation, we do not have annotations regarding **FPs** and when **SQLMap** is unable to exploit a vulnerability, we may question if it is a **FP** reported by the **SA** or just a failure of **SQLMap**. In fact, it is rather easy to confirm a vulnerability, but much harder to confirm that the software is safe in all cases [211]. Anyway, we wanted to have some assurance that the non exploited cases were **FPs** and, therefore, not exploitable or not trivial to exploit. This way, we analyzed the code where the **FPs** were located and removed the security protections of the vulnerabilities. Then, we re-run **SQLMap** to exploit the modified code. In case of success, we considered that the removed protections were, in fact, effective and that **SA** result was a **FP**. This was how we confirmed the results presented in Table 6.6. For example, 35 **FPs** in the plugin `sermon-browser.0.43` are due a type cast “(int)” of **GET/POST** parameters. Listing 6.11 provides an example of a type cast for preventing **SQLi** vulnerabilities. After removing the type cast “(int)” in line 1333, **SQLMap** reports that the parameter “mid” was successfully exploited. This example contains three **SQL** statements in different **LOCs** using the variable `$id`. To know if all **LOCs** are vulnerable requires the inspection of the database before and after of the injection to detect changes in the target records of the **SQL** statements.

Table 6.6.: FPs Tested with Indirect Call.

Plugin	SA	Our methodology						
		<i>Crawl</i>	<i>S</i>	<i>I</i>	<i>D</i>	<i>U</i>	<i>M</i>	<i>T</i>
collision-testimonials.3.0	1	1				1		1
contus-hd-flv-player.1.3	2	2				2		2
contus-video-gallery.2.8	2	2				2		2
flash-album-gallery.2.55	9	9					9	9
forum-server.1.7.1	4	4		1		1	2	4
ip-blacklist-cloud.3.4	5	4		4				4
levelfourstorefront.8.1.14	40	35	29	4		2		35
odihost-newsletter-plugin	1	1		1				1
sermon-browser.0.43	35	35	5	8	3	6	13	35
slider-image.2.6.8	10	9					9	9
wp125.1.5.3	1	1					1	1
wpforum.1.7.8	4	4				4		4
Other (5)	8							
Total (18)	122	108	34	18	4	18	34	108

S-Select, I-Insert, D-Delete, U-Update, M-Multiple SQL, T-Total.

```

...
1329 } else { // edit
1330     //Security check
1331     if (!current_user_can('edit_posts'))
1332         wp_die(__('You do not have the correct permissions to edit sermons', $sermon_domain));
1333     $id = (int) $_GET['mid'];
1334     $wpdb->query("UPDATE {$wpdb->prefix}sb_sermons SET title = '$title', preacher_id = '$preacher_id', datetime
        = '$date', series_id = '$series_id', start = '$start', end = '$end', description = '$description', time = '
        $time', service_id = '$service_id', override = '$override' WHERE id = $id");
1335     $wpdb->query("UPDATE {$wpdb->prefix}sb_stuff SET sermon_id = 0 WHERE sermon_id = $id AND type = 'file'");
1336     $wpdb->query("DELETE FROM {$wpdb->prefix}sb_stuff WHERE sermon_id = $id AND type <> 'file'");
1337 }
...

```

Listing 6.11: Slice of code from the file `admin.php` of the `sermon-browser.0.43` plugin.

6.3.4. Comparison with Alternative Approaches

By combining the results of Table 6.3 and Table 6.6, we have the global outcome of the experiments applying our methodology to obtain a confirmation of the truthfulness of the vulnerabilities discovered by the **SAST** tools. To understand the relevance of these results, we compared them with those from a scenario using only **SA** and another scenario where we have the results of the **SA** being fed directly to **SQLMap** (Table 6.7):

- 1) **SA**. Having only the results of the **SA**, there are 602 (80+382+18+122) vulnerability alarms, but none confirmed, since no other mechanism was used for verification.
- 2) **SA+SQLMap**. In this setup, the result of the **SA** is used to feed the **SQLMap** with the **URL**, vulnerable variables and other parameters required for triggering the **SS**. These data was collected from the vulnerable code reported by the **SA**. Since this is the only information available, we can only consider the confirmation of 45 vulnerabilities of the direct calling group: 27 (total of column B in Table 6.3) + 18 (total of column **SA** in Table 6.5).

- 3) **Our methodology.** The **SAST** tools are executed and the result, along with the web application interaction data, are fed to the **SQLMap** applying our methodology. This gives us 462 confirmations of the **SA** results (70+266+18+108, data from tables 6.3 to 6.6) .

Table 6.7 shows a comparison of the effectiveness of the three scenarios regarding their ability to confirm if the vulnerabilities are really exploitable. Column “SA stage” shows the number of vulnerabilities detected by the **SAST** tools. Column “Crawling stage” shows the vulnerabilities that were executed during the crawling process. Column “DA stage” has the results of using the **SQLMap** after feeding it with **SA** data. The last column, “Manual review”, depicts the number and percentage of **SA** vulnerability results that must be checked if we want to be sure that they are really exploitable. As expected, using only **SAST** tools, all the results need to be confirmed manually. This is the baseline. If we add the **DA** tool to the equation, the need for manual analysis drops to 85.4%. However, if we consider our approach, this number drops to just 23.3%, which is a huge improvement.

Table 6.7.: Comparison of the Effectiveness of our Blended Methodology.

Approach tested	SA stage	Crawling stage	DA stage	Manual review	
SA	602	-	-	602	100.0%
SA+SQLMap	602	-	45	575	93.2%
Our methodology	602	397	462	140	23.3%

The crawling process to cover part (397/602, see Table 6.7) of the results of the **SAST** tools produced many **HTTP** requests (4,571). Each **HTTP** request has one or more parameters and respective values. Without data from **SA**, we have to use **DA** for testing all **HTTP** request for all parameters of the request. This means that the number of tests to be performed will be several times the number of **HTTP** requests, as it is common to have **HTTP** requests with several parameters and each parameter should be tested individually. Furthermore, without knowing details about the structure of the **SQL** query reaching the **SS**, we have to run **SQLMap(DA)** with options for trying more combinations of **SQLMap** parameters and values. Therefore, using **SA** data tremendously reduces the efforts in crawling and **DA** processes and provides information the coverage of the **SS** executed during the crawling process.

Let’s take as example a scenario of using our crawling stage considering all **SSs** in the source code instead of the **SSs** reported by **SA** stage. In this scenario (our crawling+**SQLMap**), the crawling process has to cover all the **SS** (4,786, as can be seen in Table 6.1) of the plugins instead of the potential vulnerable **SS** (602) reported by the **SA**. Considering 12 (4,571/397) the average number of **HTTP** requests generated during the crawling process to cover one **SS**, this would be generated 57,432 (4,786x12) **HTTP** requests with one or more parameters to cover all the **SSs**, therefore increasing twelve times the effort with crawling and **DA**. Note that, using our crawling+**SQLMap** we can discover vulnerabilities not reported by the **SA** (e.g., vulnerabilities found in a third-party **API** that could not be detected by **SAST** and would require **DAST**).

Overall, the results highlight the relevance of **DA** to improve the confidence on the output of the **SA**, and also that deep knowledge about the user interaction is able to increase this even further. Without using the necessary crawling data to help configure the **DA**, a whole class of applications that have features that cannot be executed directly through a **URL** would be left

untested. In fact, the precise data required to execute a specific potential vulnerability simplifies a lot the process of [DA](#) to produce evidences that the vulnerability exists and can be exploited.

6.3.5. Threats to Validity

There are some potential threats that may affect our study and the results. These threats are as follows:

- **Dataset.** Despite the results being obtained from a relevant number of WordPress plugins with different sizes and functionalities, they only target [SQLi](#) vulnerabilities and cannot necessarily be generalized to all web applications and classes of vulnerabilities. Future studies should include both other kinds of web applications and classes of vulnerabilities.
- **Crawling.** Our approach requires crawling the applications. Thus, the results are limited to how successful this task is performed. The use of automated tools minimize the burden of this process, however they need to be carefully configured in order to execute all the vulnerable code. However, in the preferred use case for our methodology, that is the use during the development of the plugin, it is expected that the developer has automated tests in place that exercise all the functions, so they should cover all the vulnerable code as well. Moreover, using our approach, we can obtain exactly the number of results reported by the [SA](#) that are being covered by the crawling task, which helps monitoring the crawling success.
- **DA tool.** We used [SQLMap](#) to confirm the [SA](#) results. Using other [DA](#) tools does not necessarily produce the same results because each tool has different capabilities and settings.
- **Unexploitability.** When applying our methodology, [SQLMap](#) was unable to exploit some vulnerabilities. However, this does not mean that these situations are not exploitable at all. [DA](#) tools have many tuning configuration options in order to be able to bypass several security measures that may be used by the application under test. Therefore, carefully tweaking the available options is crucial for the success of the tool. Although all these options may be very powerful, sometimes a vulnerability is only exploitable in a concrete state of the application, like having specific data in the database or having a PayPal account. In these cases, a deep knowledge on the inner workings of the application may be necessary to properly configure the tool. To confirm the results, we manually investigated all the cases where [SQLMap](#) was unable to exploit the vulnerabilities. In these cases, we used the same exploit that was used by the [SQLMap](#) but we could observe the database change by looking at it directly, so our conclusions were based on the analysis of more comprehensive data.

6.4. Conclusion

In this chapter, we presented a blended approach using [SA](#), [DA](#) and application interaction for vulnerability detection. After running the [SA](#), the target application is executed while its interaction is being recorded so it can provide the [DA](#) with the necessary intelligence to increase

its detection capabilities. The outcome of this methodology is a framework that is capable of discovering a large set of proven to be exploited vulnerabilities in web applications.

The experimental evaluation considering a set composed by a population of 602 SA results (462 SQLi TPs and 140 FPs) from 49 WordPress plugins, show that the proposed approach was able to outperform by far current procedures using only SA to search for vulnerabilities. In fact, it was able to confirm most of the SA results as being either TPs or FPs, in a way that it shortens the number of the results that need to be checked manually to less than 1/4. Furthermore, the process can be performed in an automated fashion, optimizing both the speed of operation and its coverage, which is great for security practitioners and penetration testers.

Future work should focus on two main directions. The first is improving the automation of the crawling, by gathering data from several sources (like unit test data, database, DA) and the use of AI algorithms to process them in order to build the necessary knowledge to fill in the form fields and follow the correct path to the vulnerability location. Another direction is the automatic verification of the cases missed by the DA. Many of these situations have to do with the difficulty of the tool to detect a successful attack. By analyzing the database flow data, obtained from the trace files, a database reverse proxy, or instrumenting the database driver, etc., it will be possible to obtain the necessary data.

Conclusions and Future Work

Web applications became the most prevalent platform used by organizations of any size to provide business information and critical services to their clients and partners. The security of such web applications is, therefore, a top priority for both business managers and clients. Given the assets at stake, it is not a surprise to see vulnerable web applications as the most common way attackers use to penetrate organizations from the web or from within the organization to gain access to sensitive information. The pressure of society and legislation rules targeting the need for security and privacy is making the web applications security a major concern of companies throughout the world. On the other side, given the pressure to quickly develop more and more web applications with more features, results in code written in a rush, with insufficient security requirements, ignoring most of the threats they will face when deployed in the real world and the means necessary to cope with them.

Several authors state that approximately half of the security vulnerabilities could be prevented by addressing them at the program code level, during development. Two of the most important vulnerabilities exploited by malicious actors are [SQLi](#) and [XSS](#). [SQLi](#) vulnerabilities are highly requested by attackers given the benefits they can obtain, since they allow attackers to bypass authentication, extract, modify and delete data on back-end databases (every website has a server, database, and other applications), take complete control of the underlying databases, servers and networks. [XSS](#) may allow attacks to deface web sites, hijack the current session of the user, manipulate or steal his cookies, and steal his identity, spying his web use, or even redirect the user to malicious web sites, infecting the user's computer with a virus or other malware, with the possibility to take full control of the machine.

PHP is by far the most used server-side scripting language to develop web applications. Just about 80% of all websites are running on PHP. Well-known websites, such as Facebook, Wikipedia, and WordPress are built using PHP as their server-side script language. WordPress powers more

than 42% of the websites today and can be extended with plugins providing features for every possible use, making developing a modern web application an easy task. But there is a dark side in these plugins. This high demand of plugins attracts many developers without the necessary skills, so we can see plugins as the source code responsible for almost all the vulnerabilities of websites developed using WordPress.

This thesis addressed the security of web applications, focusing on [SQLi](#) and [XSS](#) vulnerabilities in WordPress plugins. The overall objective was to propose new and more effective techniques and tools to detect the most important security vulnerabilities in web applications by automatically blending Static Analysis ([SA](#)) and Dynamic Analysis ([DA](#)) and use this as an advancement in the state of the art of web application security. This goal was achieved with the contribution to increase the knowledge about benchmarking [SAST](#) tools for vulnerability detection and with the proposal of methodologies that benefit from this knowledge to help providing more secure web applications. Throughout the course of the research, a number of dissemination activities was carried out, resulting in two international conference papers and three papers in international journals.

7.1. Key Contributions

Regarding our research work, this thesis presented the following key scientific contributions:

- 1) **Development of a PHP [SAST](#) tool for [SQLi](#) and [XSS](#).** We developed a tool ([phpSAFE](#)) for detecting [SQLi](#) and [XSS](#) vulnerabilities in PHP plugins, including those developed using [OOP](#). [phpSAFE](#) is configured out of the box to verify the source code of WordPress plugins without the need to have WordPress installed. The evaluation results show that [phpSAFE](#) clearly outperforms other free [SAST](#) tools. The output of the vulnerability detection is presented in a web page that helps review the results, including useful data for developers to quickly fix the vulnerabilities reported. [phpSAFE](#) is also prepared to be easily integrated in developer workflows. For example, the use of [phpSAFE](#) can be part of the [SDLC](#) of a software development company, it can be used to automate the process of analyzing a large quantity of PHP scripts residing in different locations, it can be tuned to produce and store the results in several formats and distribute them over the network. This tool can be used by both occasional developers and professional software houses wanting to speed up the development process of more secure software and reducing costs avoiding the use of expensive commercial [SAST](#) tools.
- 2) **Development of a methodology for benchmarking [SAST](#) tools for web application security.** [SA](#) is one of the most important activities to discover vulnerabilities in the early stages of the [SDLC](#). Unfortunately, [SAST](#) tools have limitations and they can fail detecting vulnerabilities ([FNs](#)) and, at the same time, they can produce many [FPs](#). Consequently, different [SAST](#) tools tend to return quite different results, and the selection of the [SAST](#) tool that best fits a specific project is a challenging task. Benchmarking could assist in the selection of alternative [SAST](#) by comparing their behavior while testing relevant applications. However, the currently available benchmarks targeting [SAST](#) tools are very limited. To fill this gap, we proposed a methodology to design benchmarks for

the evaluation of **SAST** tools that detect vulnerabilities in web applications considering different levels of criticality. The approach for the definition of the benchmarks proposed consists in the specification of the benchmark components: scenarios, workload, metrics, and procedure and rules. By exploring the notion of *application scenarios* (a scenario is a realistic situation of vulnerability detection that depends on the criticality of the application being tested and on the security budget available), our approach allows a better match of its outcomes with the environmental requirements for the **SAST** tool operation. To compose the workload, we consider four scenarios (highest-quality, high-quality, medium-quality and low-quality) and a representative group of vulnerable applications for each scenario. This assures that the **SAST** tools are tested considering the need to address both the complexity and the way real code is built, instead of processing much simpler synthetic code samples or test cases (as done by existing **SAMATE** and **BSA** benchmarks). Our approach relies on one *main metric* and a *tiebreaker metric* for each scenario. The main metric is used to rank the **SAST** tools and the tiebreaker metric is used to decide eventual ties between two or more **SAST** tools.

- 3) **A benchmarking campaign of SAST tools for web security.** We conducted a case study to demonstrate the validity and applicability of our benchmark methodology by benchmarking five free **SAST** tools detecting **SQLi** and **XSS** vulnerabilities in 134 WordPress plugins. The experimental results showed that the best tool changes from one scenario to another and also depends on the class of vulnerabilities being detected. The comparison of the results using our metrics and the metrics from **SAMATE** and **BSA** reveals that the use of the same metrics for all scenarios makes it more difficult to choose the most appropriated tool for a project with specific security requirements. Given its performance, our novel benchmark approach is a valuable tool to help project managers choosing the best **SAST** tools according to their needs and the resources available.
- 4) **Case studies combining the results of five SAST tools detecting SQLi and XSS vulnerabilities.** **SAST** tools provide some capabilities with proprietary and limited functionality, with each tool providing value only in subsets of the enterprise application space. To overcome this limitation, developers may need to use more than one **SAST** tool in order to combine their strengths and avoid their weaknesses. The use of multiple **SAST** tools might be helpful, as more vulnerabilities are likely to be reported, however, the drawback is that the number of **FPS** may at the same time increase. Furthermore, the acceptable/expected outcome of the static analysis process (in terms of coverage and **FPS**) depends on the development scenario. As there is a lack of studies combining the results of several **SAST** tools in workloads composed by real applications annotated in terms of vulnerabilities and non-vulnerabilities, we conducted an experiment to study the potential of combining the outputs of multiple **SAST** tools as a way to improve the performance of vulnerability detection across different realistic development scenarios. Our study was based on the results of the benchmark mentioned above. The results showed that combining the outputs of several free **SAST** tools does not always improve the vulnerability detection performance. Thus, the best solution may be a single tool or a combination of tools that may not include all the tools under evaluation. A key observation is that there are cases where using a single **SAST** tool provides better results than combining multiple tools. In principle, combining multiple **SAST** tools has benefits due to the complementary of their

results. However, for solutions including **SAST** tools that report many **FPs**, the overall performance is worse in some scenarios. In general, as the number of **SAST** tools in a combination increases, both the number of new vulnerabilities found and the new **FPs** reported increase less and tend to stabilize. In fact, our results highlight that, the best combination of **SAST** tools is highly dependent on the specific situation, and it should be selected after a properly targeted benchmarking procedure, such as ours. By using our methodology, a developer is able to choose which is the best combination of **SAST** tools that fits better in the project requirements.

Different **SAST** tools analyzing the same code report different sets of vulnerabilities and different sets of **FPs** with some overlap. We conducted an in-depth analysis of the code of several WordPress plugins to find reasons to justify why some tools do not detect vulnerabilities that other tools detect. Based on the code of the plugins, we created several test cases with **OOP** and/or **POP** code with **SQLi** and **XSS** vulnerabilities. We ran the tools for detecting vulnerabilities in the testes cases. We found several reasons why the **SAST** tools may fail to detect vulnerabilities (for example, the way in which they analyze arrays, control flow constructs). The results also highlights where their developers should look in order to fix them.

- 5) **Proposal of a methodology blending static and dynamic analysis for **SQLi** vulnerability detection in web applications.** Current approaches based only on the results of **SA** to feed the **DA** have the limitation of not providing enough data to exploit all the vulnerabilities discovered. Our methodology combines **SA**, a crawling procedure and **DA** to automatically generate a set of specific inputs and configuration options to guide with efficiency and efficacy the **DA** in the process of successfully exploiting each vulnerability reported by the **SA**.
- 6) **A case study using the proposed blending of static and dynamic analysis for **SQLi** vulnerability detection in web applications.** The proposed approach was evaluated using a large number of **SQLi** vulnerabilities in WordPress plugins. Our approach was able to confirm either as a vulnerability or a false alarm over 3/4 of the results reported by the **SA**, decreasing tremendously the usual need for manual analysis, which is a huge improvement for security practitioners. In fact, by using our approach it is possible to identify and work on the fix of the vulnerabilities (while spending the least possible amount of resources dealing with false alarms), as it provides lots of data related with the vulnerabilities, including the command used to exploit the vulnerability (a **PoC** that the vulnerability exists and can indeed be exploited). All the data provided are automatically correlated, to get a clearer view of the vulnerabilities from the source code of the web applications to the exploit of instances of these web applications running. Having the right data in place is key to understand where there might be vulnerable code in the web application and to improve it quickly before a vulnerability becomes a breach. For instance, even security unskilled developers can test the suspected vulnerability by running the command, fix it using the data provided (e.g., file, line, vulnerable variable, **EPs** and **SS**) coming from the **SA**, test the fix using the **PoC**, and have a documented proof that the issue has been probably resolved.

It is surprising how many options are out there for improving web application security. Our methodologies and case studies provide very interesting and quite valuable results that can

contribute right away to improve important aspects of web application security. Our outcomes are a great place to start. This is the case of an exploratory study on Machine Learning to combine security vulnerability alerts from [SAST](#) tools [212] conducted by D'Abruzzo Pereira et al. In practice, based on our dataset they used four Machine Learning (ML) algorithms, alerts from two [SAST](#) tools, and a large number of Security Misconfiguration to predict whether a source code file is vulnerable or not and to predict the vulnerability category. Results show that one can achieve either high precision or high recall, but not both at the same time. By analyzing and comparing snippets of source code, they demonstrated that vulnerable and non-vulnerable files share similar characteristics, making it hard to distinguish vulnerable from non-vulnerable code based on [SAST](#) tools alerts and [SMs](#).

It is important to emphasize that, although we focused on the top two web application vulnerabilities, [SQLi](#) and [XSS](#), and on the most popular programming language for web development, [PHP](#), our methodologies can not only be extended to other vulnerabilities but also be applied to other scripting languages like [Python](#), and programming languages like [Java](#) and other technologies.

7.2. Future Work

Although the work presented in this thesis has yielded significant research results, these topics still present avenues of research that are worthy of further pursuit. In this section, we highlight some possible research topics that we consider to be relevant as a continuation of the work presented in this thesis:

- 1) **Enhancement of the proposed [phpSAFE SAST](#) tool.** [phpSAFE](#) was designed and implemented for [PHP](#) and it is able to detect [SQLi](#) and [XSS](#) vulnerabilities in [PHP](#) code. The vulnerability detection output should be enhanced with more attributes, like the [OWASP](#) and the [CWE](#) vulnerability classes, as well as using other formats, like [CSV](#) and [XML](#). The adoption of other formats helps tremendously reducing the complexity of the integration/merging of the results with other tools. The benchmarking results presented reveal that the tool fails the detection of basic vulnerabilities in some specific situations. Using the detailed information resulting from our detailed analysis of the source code, the tool could be improved to detect these vulnerabilities. Another enhancement is to extend the tool for detecting other classes of vulnerabilities, like Local File Inclusion ([LFI](#)), Remote File Inclusion ([RFI](#)) and OS Command Injection ([OSCI](#)).
- 2) **Create new instantiations of the benchmark approach for [SAST](#) tools and make them public.** Benchmarking results help project managers choosing the best [SAST](#) tool according to their needs and the resources available. Benchmarking workloads that are fully characterized both in terms of vulnerabilities and non-vulnerabilities are valuable resources for researchers to perform other studies and perhaps devise new methodologies to improve vulnerability detection in web applications. Developers can extend the benchmarking by adding other existing tools, new versions of these tools and new emerging tools. Doing this periodically, allows developers to have a picture of the state of the art of the tools facilitating their choice for each specific project. Based on the locations of the vulnerabilities in the source code, [SAST](#) tools developers can easily investigate why the tools fail and how

to improve these situations. Emerging benchmarks, like those we presented, need to gain the confidence of the community in order to be accepted. This is a huge step towards their wider adoption allowing the community to provide important feedback about their use in situations where other **SAST** tools are used.

- 3) **Combining results of several SAST tools.** Further studies should be performed to investigate optimal adjudication setups to improve both the sensitivity and specificity of **SAST** tools. For example, to investigate different strategies of combining the **SAST** tools, such as *k-out-of-n*. Other direction is to assign a security risk like a Common Vulnerability Scoring System (**CVSS**) to the vulnerabilities reported by the **SAST** tools and use this additional information to combine the results of the **SAST** tools. One of the outcomes of this thesis, is that the effectiveness of the **SAST** tools depend on the type of constructs used in the source code of the web applications. Therefore, other possibility to study is the use of Machine Learning algorithms based, for example, in security misconfigurations and language constructs, in order to learn in detail the types of code constructs that the **SAST** tools are able, and not able, to inspect to detect vulnerabilities. This knowledge (e.g., tool *X* is not able to analyze PHP associative arrays) can be used to combine the **SAST** tools in order to cover all types of code constructs in the source code.
- 4) **Studies to identify strengths and weaknesses of SAST tools.** We concluded that using small test cases derived from the source code of vulnerable applications is a helpful way to find strengths and weaknesses of the **SAST** tools. Therefore, the process for extracting the slices of code from the source code and derive test cases should be automated. With this achievement it would be possible to generate test cases based on real source code, including types of current code used in the development of web applications.
- 5) **Improve the blending of static and dynamic analysis for vulnerability detection.** Improving the automation of the crawling, by gathering data from several sources (like the unit test data and the database) and improving the automated verification of the cases missed by the **DA**, which can be done using several **DA** tools as each one produces different results, will lead to better results. A key aspect is that, even using many **SAST** tools, there are vulnerabilities that remain undetected. Therefore, from all **SSs** in the code, the **SA** may report only a subset of these **SSs** as potentially vulnerable. Our crawling process captures runtime data related with these **SSs** and the **DA** exploits these **SSs**. One direction to improve the vulnerability detection is to consider also as potential vulnerabilities the **SS** executed during the crawling process and not reported by the **SA** and then proceed with the **DA** process as in our proposed methodology. It is important emphasize that without data (e.g., **EPs**) from **SA** it is more difficult to exploit the **SS**. The **EP** provides the parameter names to be tested by the **DA**. For instance, the **DA** have to test all parameters in the **HTTP** requests to the web application.

Assigning Applications to Scenarios

This appendix provides details about the stage [Assigning Applications to Scenarios](#) in Section [4.1.3.2](#).

The organization of the appendix is as follows. Section [A.1](#) presents background about quality models for measuring the internal quality of software products. Section [A.2](#) presents the process for assigning applications to scenarios.

A.1. Characterizing Software Quality

Several software quality models were proposed and many tools were created to control the development and maintenance of software [\[213\]\[214\]\[195\]\[215\]](#). Among others, these tools are used to identify problems in the source code early in the development process, allowing project managers to take mitigation actions. In fact, several studies show that there is a relation between the quality of the source code and the failures of software products [\[216\]](#). For example, it is known that code units that have the highest complexity also tend to contain more defects [\[217\]](#).

Web applications have common characteristics of traditional software, however they also have unique characteristics that are related to the distributed nature of the Internet. The use and reuse of third-party components developed in multiple languages, the use of web interfaces, the speed of access to data, and the security of transactions [\[218\]](#). Thus, traditional software quality models may not be adequate to fully assess the quality of web applications. Since web applications became an indispensable platform in all sectors of our society, researchers proposed models for assessing the quality characteristics of web-based applications. Nabil et al [\[219\]](#), proposed a software quality model for web-based applications that extends the ISO 9126 software quality model by adding characteristics such as reusability, scalability, credibility, security, popularity

and profitability, among others. They organized these characteristics in three views: developers, owners, and visitors. Sankar et al. [218] proposed common quality attributes for secure web applications organized in four quality categories: design, run-time, system, and user.

Several Source Code Metrics (SCMs), like the CCN2 (a variation of the CCN adapted for OOP [220]) [182], or the number of LLOC¹, have been proposed to measure quantitatively the quality of software products [222]. It is common to organize the code of an application across several files, functions and classes. Thus, for each SCM may results several measures according to its code organization. Averaging these results to calculate a single value for a SCM is fundamentally wrong because each measure may represent different amounts of code in the applications. A common method for aggregating the measures of a SCM is to build a risk profile (see Section A.2.2.2-Risk Profiles for more details) based on a set of predefined thresholds for the SCM [223]. This allows developers to focus on software units where SCMs are exceeding the thresholds first and the others later, as units with higher values for several SCMs tend to have more faults [216].

An appropriate use of SCMs requires risk thresholds to determine whether the value of a SCM is acceptable or not. These risk thresholds vary widely in the literature. For example, the limit of 10 for CCN was proposed by McCabe [220], but limits as high as 15 have also been used successfully [224]. In fact, the risk thresholds are defined based on the opinion of software quality experts for particular contexts [225]. For example, in high quality software, it is admissible to have small percentages of source code with high values for some SCMs to express a balance between real needs and idealized design practices [225].

Alves et al. [222] proposed a methodology for deriving risk threshold values for SCMs, based on data analysis from a representative set of applications. This methodology has been successfully used in several works. One example, there is the method proposed by Baggen et al. [181] to rate the maintainability of the source code of applications (from 0.5 to 5.5 stars) based on risk profiles and a set of rating thresholds. The Baggen et al. method is applied by the Software Improvement Group (SIG) to annually re-calibrate its quality model [226], which forms the basis of the evaluation and certification of software maintainability conducted by SIG [188] and TÜViT [181].

The Static Analysis Community has recognized that the analysis of source code is harder than it is usually assumed [180]. The participants of the NIST workshop on Software Measures and Metrics to Reduce Security Vulnerabilities (SwMM-RSV) recommended that code should be amenable to automatic analysis [180]. Therefore, the analyzability (sub-characteristic of maintainability) should be measured and increased to make the code readily analyzable. This contributes to reduce vulnerabilities, as tools tend to perform better in less complex code.

A.2. Process for Assigning Applications to Scenarios

The web applications collected to compose the workload have to be assigned to scenarios. Our methodology to assign each web application to a scenario is based on the **Internal Quality** of all web applications in the workload. Figure A.1 depicts the overall process for assigning

¹LLOC - the number of instructions or statements in an application [221].

applications to scenarios. The gray boxes represent the main steps detailed in the next sections.

The organization of the next subsection is as follows. Section A.2.1 details the quality model used in our benchmark approach. Section A.2.2 details the process for gathering source code metrics. Section A.2.3 details the process of deriving ratings of applications.

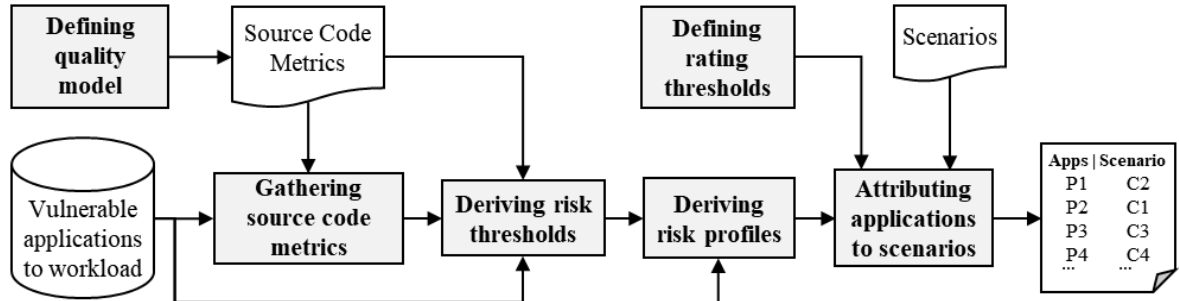


Figure A.1.: Process for assigning applications to scenarios.

A.2.1. The Quality Model

Product quality, along with process quality, are two of the most important aspects of software development nowadays. There are several quality models for mapping software metrics onto a quantitative quality. Examples are: the ISO/IEC 9126 Software Product Quality [227], the Quality Model for Object Oriented Design (QMOOD) [187] which makes use of quality characteristics defined in the model ISO/IEC 9126, the Software Quality Assessment based on Lifecycle Expectations (SQALE) [228] based on the Boehm model [229], McCall [230], ISO/IEC 9126, and more recently the ISO/IEC 25000 Systems and Software Quality Requirements and Evaluation (SQuaRE) [231], which is the result of blending the ISO/IEC 9126 and ISO/IEC 14598 series of standards.

The main objective of SQuaRE is to assist in the development and acquiring of software products through the specification of quality requirements and the evaluation of quality characteristics [231]. The product quality model defined in ISO/IEC 25010 comprises eight quality characteristics shown in the figure A.2. The SQuaRE model organizes the model in three levels. In the first level, it defines a set of quality characteristics (e.g., **security**, **usability**, and **maintainability**). In the second level, for each characteristic it is defined a set of sub-characteristics that contribute for the quality of characteristic (e.g., **modularity**, **reusability**, **analysability**, **modifiability**, and **testability** contribute to maintainability, see figure A.3). In the last level, there are defined a set of Software Product Properties (SPPs) (e.g., function complexity) which are connected to the sub-characteristics. To measure the SPPs are used the SCMs (e.g., cyclometric complexity). As an example, the cyclometric complexity is used to measure the complexity property, which contributes for the testability and changeability sub-characteristics. These sub-characteristics contribute for the maintainability characteristic and finally for the overall software product quality.

The SQuaRE product quality model defines the relationships between the quality characteristics and quality sub-characteristics and leaves the task of choosing the SPPs and respective SCMs to

the user. For instance, the user can choose the characteristics that fits his quality goals. Figure A.3 shows an example where the user chose only the characteristic of maintainability and the related sub-characteristics based on the ISO/IEC 9126 model that are widely accepted both by industrial experts and academic researchers.



Figure A.2.: Product quality model defined in ISO/IEC 25010. Adapted from [231].

In the context of this work, we are interested in measuring the internal quality of the software, which is done without running the software, and it is based on a set of SCMs. The goal is to assign a scenario to applications. This means that these applications could be used in this scenario of criticality because their source code has sufficient target quality.

At first glance, internal quality does not matter to users and customers since internal quality is not something that customers or users can see. In fact, users do not want to pay more for similar software with the same functionalities but with different internal quality. Users are likely to pay more to get a better user interface and user-experience (i.e., external quality). Therefore, many software developers do not put their time and effort into improving the internal quality of their software. Moreover, usually there is a high pressure put into developers to quickly deliver new functionalities, leading many of them to complain that they don't have time to work on architecture and code quality [232].

A fundamental role of internal quality is that it lowers the cost of future change. Programmers spend most of their time modifying code. Even in a new system, almost all programming is

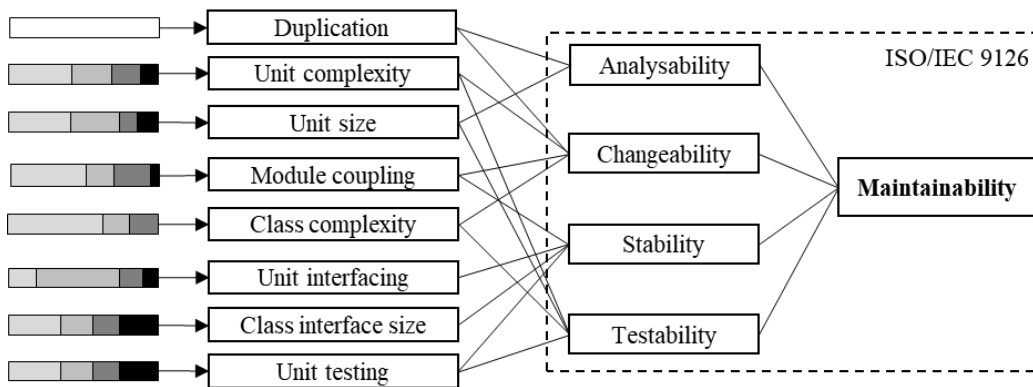


Figure A.3.: Relation between SPP and software product sub-characteristics of maintainability (image adapted from [222]).

done in the context of an existing code base. Therefore, high internal quality makes it easier to enhance software and allows a team to add future features with less effort, time, and cost. This means that putting the right time into writing good code actually reduces the overall cost. In fact, writing software with good internal quality in the short term requires some extra efforts and costs. However, a long term high quality software is quicker and cheaper to produce, maintain and extend [232].

There are several internal quality models for assessing, comparing and certificating the quality of software products. In this work, we used the methodology proposed by Alves et al. [188] for building a maintainability quality model. This approach uses a standardized measurement model based on the ISO/IEC 9126 definition of maintainability and it is detailed in the next paragraphs.

The ISO/IEC 25040:2011 (SQuaRE-Evaluation process) contains guidelines for the evaluation of software product quality [231]. It provides a process description organized in four stages:

- 1) **Establish evaluation requirements:** this stage includes three steps:
 - a) *Establish the purpose of the evaluation.* In the context of this work, the proposal is to evaluate the internal maintainability of the source code of web applications.
 - b) *Identify types of products.* The types of products are web applications developed using **POP** and/or **OOP** programming language paradigms.
 - c) *Specify quality model.* The quality model is based on the model proposed by Baggen et al. [181]. The approach uses a standardized measurement model based on the ISO/IEC 9126 definition of maintainability and a set of source code metrics (**SCMs**). The software product properties (**SPPs**) chosen are at application level; complexity and coupling at class level; size, testing, interfacing, and complexity at function/method level as shown in the Figure A.3. The desired quality indicator for the applications is a quality rating from 1 to 5. The number 5 represents the best maintainability score and the number 1 the worst. Besides expressing source code maintainability in terms of numerical values, the model also provides meaningful results, i.e. it gives a detailed list of source code fragments that should be improved in order to reach an overall higher quality [233]. For example, the code of a class's method with high complexity (e.g., **CCN2** > 30) will be highlighted as code that should be improved.
- 2) **Specify the evaluation.** It includes three steps:
 - a) *Select source code metrics.* the proposed **SCM** for evaluating the chosen product properties are listed in the Table 4.3. It includes the scope (i.e., level of application, class and unit) and description. Unit is the smallest piece of code that can be executed and tested individually, for example a **Java** method or a **PHP** function.
 - b) *Establish rating levels for source code metrics.* After selecting the source code metrics to be used, it is necessary to establish the scale and meaning of the score. For all source code metrics, we proposed a numeric scale in the admissible ranges of the metric (e.g., **CCN2** [1, $+\infty$]). The score of the selected metrics increase as their quality decreases.
 - c) *Establish criteria for assessment.* In this step are defined the aggregation model and the model to map the aggregated data into the desired quality ratings. For the aggregation model, we proposed **risk profiles** (several values of the measurements of the **SCMs** in a single value) for the metrics at unit and class levels (e.g., **CCN2**) and

a grand total (i.e., one measure for the entire web application) for **duplicated line density** (See Figure A.3).

The aggregation model is based on the method defined by Alves et al. [222] (Section A.2.2.1) for assessing the maintainability of software. This method requires a set of representative applications in terms of source code for deriving the **risk thresholds** used in the calculation of the risk profiles. Since our workload has to be representative of all applications in a specific domain, we propose the web applications in the workload as the set of representative applications in terms of source code.

The risk profiles are then mapped in five quality ratings. The mapping method used is based on the method also proposed by Alves et al. [223] (Section A.2.2.4) and requires a set of **rating thresholds** for each SCM.

- 3) **Design the evaluation.** this stage produces an evaluation plan. It includes the selection of the tools and configuration settings for gathering automatically the measures of the select metrics in 2), and the identification of the applications including the list of files to be analyzed and a list of files to be excluded from the evaluation and why (e.g., third-party or test class).
- 4) **Execute the evaluation.** The evaluation is carried out following the procedures and plan defined in the previous stages.
 - a) *Take measures.* In this step, the tools for gathering the measures of the SCMs are executed and the results stored. Then, as defined in 3), the results of the metrics at the unit level are aggregated in risk profiles. The risk profiles are then compared against the rating thresholds for determining the quality sub-ratings for each SCM. The ratings of all SCMs are averaged in order to calculate the quality rating for each sub-characteristic of maintainability. Finally, the sub-characteristics ratings are averaged to provide the overall quality rating for the maintainability of the application.
 - b) *Compare with criteria (rating).* The quality rating is mapped to a scenario (1 to 4, See Session 4.1.1). Since the ratings vary from 1 to 5 in ascendant quality and the scenarios from 1 to 4 in descendant level of criticality, we used a simple mapping rating-scenario: 1-4; 2-4; 3-3; 4-2; 5-1 (see Table 4.5). We joined the ratings 1 and 2 in the scenario 4, less stringent, and a mapping 1 to 1 for the more rigorous scenarios.
 - c) *Assess the results.* The results should be compared with the expected results, when there is available information, to compare the scenario assigned to the application with the real scenario where the application is being used.

Table A.1, outlines the selected product properties and their relationship with the sub-characteristics of the maintainability characteristic. The table also illustrates an example. The sub-characteristic average rating is obtained by averaging the ratings of the properties where a the '×' is present in the sub-characteristic's line in the table. The final rating is obtained by adding the average ratings and dividing by 4, in the example shown in the table: $(4.0 + 4.0 + 2.6 + 3.5)/4 = 3.5 \approx 4$ stars.

Table A.1.: Mapping of SPPs to ISO/IEC Sub-Characteristics of Maintainability and an Example.

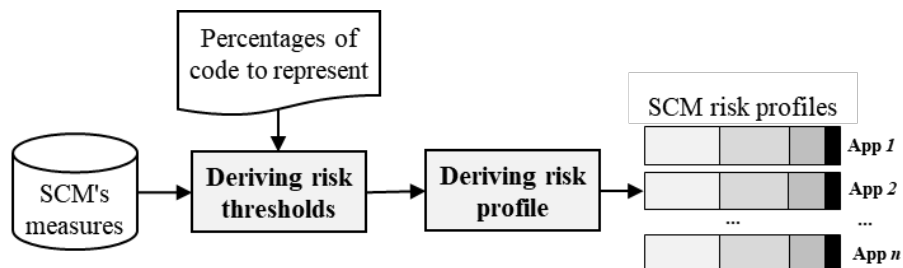
Sub-characteristic	SPPs								Average Rating
	Duplication	Unit complexity	Unit size	Module coupling	Class complexity	Unit interfacing	Class interface size	Unit testing	
Rating example	5.0	4.0	3.0	4.0	3.0	3.0	2.0	3.0	
Analyzability	×	×	×						4.0
Changeability	×	×		×	×				4.0
Stability						×	×	×	2.6
Testability		×	×	×				×	3.5
Maintainability rating (average: ★★☆☆)									3.5

A.2.2. Gathering the Source Code Metrics

There are many tools for gathering the source code metrics (e.g., PDM and FindBugs for Java, PHPdepend for PHP). Therefore, we have to select a tool or tools able to measure the SCMs defined in the quality model at the defined levels (e.g., application, class and unit (method/function)). Then, run the tools, “Execute the evaluation”, “Design the evaluation” and store the results (Section A.2.1).

A.2.2.1. Deriving the Risk Profiles for the SCMs

Figure A.4 depicts the overall process for deriving risk profiles for SCMs. First, are obtained the *risk thresholds* for the SCM based on the *percentages of code to represent* (e.g., 70, 80, 90), and on all measures of the SCM of all applications in the workload. Then, using the risk thresholds it is calculated the risk profile of the SCM for each application in the workload.

**Figure A.4.:** Process for deriving risk profiles for SCMs.

An application may be composed of several units (e.g., Java method or a C function). To measure the complexity at the unit level, we can use the CCN2 metric [220]. However, for an application, we can easily generate several thousands of measurements for the CCN2 due the high number of functions, classes and methods that compose the application. This raises the problem of knowing what is the complexity of the overall application. Therefore, to calculate the

value of the **CCN2** for a class (composed by several methods each one with its own complexity) or for the entire application (composed by several classes), we need an **aggregation function** (e.g., average, sum) to determinate the value of the **CCN2** at these higher levels (class or entire application). Aggregation by averaging is fundamentally flawed because the distribution of many SCMs follows a power law distribution, which is heavy-tailed [216]. For instance, for an application with 5 methods with **CCN2** 10 and one method with **CCN2** 100, the average **CCN2** is $30 = ((5*10+100)/(5+1))$ and for an application with 6 methods with **CCN2** 30, the average **CCN2** is also $30 = ((6*30)/6)$. Thus, these two applications should have the same average **CCN2**. However, the risk of the source code of the applications is very different, because it hides the existence of a method with very high **CCN2** for the first application.

A.2.2.2. Risk Profiles

A common method for aggregating many values of a SCM measures in a few number of values, is to build a **risk profile** based on some predefined **risk thresholds** for the SCM [223]. For example, to determine the risk profile to characterize the code in four **risk categories** (i.e., percentage of code in low, moderate, high, and very high risk) we need three risk thresholds. As an example, based on the risk thresholds 10, 20 and 50 for **CCN2**, we are able to define four risk categories, following a similar categorization of the Software Engineering Institute, as indicated in Table A.2. For example, low risk ($\text{CCN2} \leq 10$), moderate risk ($\text{CCN2} \in]10, 20]$), high risk ($\text{CCN2} \in]20, 50]$), and very high risk ($\text{CCN2} > 50$),

Developers could focus on modules where SCMs are exceeding the higher thresholds first and the others later. In fact, software modules with higher values for many SCMs tend to have more faults [216]. Moreover, fixing issues in these modules takes several times more time [234]. For example, for systems rating 4 stars, issues were found to be resolved 3 times faster than for systems rating 2 stars.

Table A.2.: Risk thresholds and risk categories for **CCN2**.

CCN2	Risk category
[1, 10]	Low risk
]10, 20]	Moderate risk
]20, 50]	High risk
> 50	Very high risk

Figure A.5 shows the risk profile for the metric **CCN2** calculated from the methods of two applications (A and B). In the figure, each color represents the percentage of LOC of the methods that fall in each of the risk categories. For instance, the application A contains 70% of code with low risk, 10% with moderate risk, 15% with high risk, and 10% with very high risk. The risk profile allows to identify problematic locations in the source code guiding individual developers in their coding tasks. Moreover, it also can be used for decision support. However, to compare applications directly based on their risk profiles it is difficult. For instance, for the applications A and B which one has better quality?

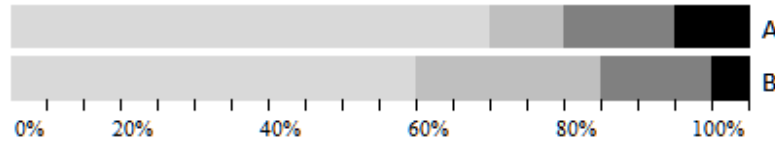


Figure A.5.: Risk profiles for methods 2 application (A, B), metric CCN2.

A.2.2.3. Deriving the Risk Thresholds

The risk thresholds used to calculate risk profiles for SCMs vary widely in the literature. For example, the limit of 10 for `CCN2` was proposed by McCabe [220], but limits as high as 15 have been used successfully as well [224]. In fact, the risk thresholds are defined based on the opinion of software quality experts in particular contexts [225]. Alves et al. [222] proposed a methodology for deriving SCMs thresholds values based on *data analysis* from a representative set of applications. This methodology has successful been used to derive risk thresholds, for example, by the Software Improvement Group (`SIG`) for software analysis [188] and for certification of technical quality of software products [226]. The main steps of the method proposed by Alves et al. for each SCM are (it includes an example):

- 1) **Metric extraction.** from the raw data of the tools used for gathering the `SCM` measures, extract for each application, and for each unit the measured value of the `SCM` and the respective number of LOCs as a weight (i.e., pairs: `SCM value:#LOC`). As an example, let's consider the method `WC_Gateway_Mijireh.process_payment()` with `CCN2` 11 and 111 LOCs (i.e., pair 11:111), in the `Woocommerce.2.3.0` WordPress plugin.
- 2) **Weight ratio calculation.** for each unit, it is calculated the weight ratio of the application, by dividing the number of LOCs of the unit by the total number of LOCs in the application. As an example, let's consider the weight ratio of the method `WC_Gateway_Mijireh.process_payment()` that is 0.00192 (i.e., 111/55527), where 111 is the number of LOCs and 55527 is the total number of LOCs of the plugin `woocommerce.2.3.0`. This method represents 0.192% of the code of the plugin with moderate risk (i.e., `CCN2` \in]10, 20]). The sum of all weight ratios will be 1.0.
- 3) **Unit aggregation.** the weight ratios of all units are aggregated by the value of the `SCM`, i.e., a histogram describing a weighted `SCM` distribution for the application. For example, there are 32 methods in the `woocommerce.2.3.0` WordPress plugin with `CCN2` 11, which represents 3.73% of all LOCs of the plugin. These values are used in 6) for determinate the risk profile for the `SCM` of the application.
- 4) **Application aggregation.** the weight ratios of all applications are normalized for the number of LOCs of all applications. Then, all units of all applications are aggregated by the value of the `SCM`, i.e., a histogram describing a weighted `SCM` distribution for all application. As an example, according to the set of applications in the workload, there are 178 methods with `CCN2` 11, which represent 2.3% of the code of all applications in the workload. So, there is in the workload, 2.3% of code with moderate risk.
- 5) **Weigh ratio aggregation.** the data from the previous step, i.e., the set of pairs value-weight, is ordered by the value of the `SCM` in ascending order. Then, it is added a third column with the accumulate weight (i.e., a Density Function for the value). To determine

the value of the SCM, which represent 60% of all code, we take the maximum value where the accumulated weight is less or equal than 0.6. So, for 60% of the overall code of the applications in the workload, the maximum **CCN2** is 12.

- 6) **Risk threshold derivation.** The risk thresholds are derived according to the percentage of code that we want to represent. For the applications in the workload, to represent 90% of the overall code for the **CCN2** of the methods, the derived risk threshold is 64. This means that 90% of the overall code has **CCN2** less or equal to 64, and at the same time that 10% of the code has **CCN2** greater than 64. To obtain the risk thresholds for determining the risk profiles for the SCM, we used the percentages proposed by Alves et al. (70, 80, 90) [222]. This means 70% for the moderate risk, 80% for high risk, and 90% for very high risk. As described in 5), using these percentages we obtain the three risk thresholds of the SCM.

A.2.2.4. Deriving Risk Profiles of the SCMs of the Applications

With the set of risk thresholds for each **SCM**, we calculate the risk profiles for each application. Therefore, the process for determining the risk profile of an application has two steps:

- 1) **Data preparation.** This step, is like the step 5) of deriving risk thresholds for SCMs (Section A.2.2.1), but for a single application, instead of all applications. Therefore, the set of pairs *value:weight* that comes from the step 3) ((Section A.2.2.1)), is ordered by the value of the SCM in ascending order. Then, it is added a third column with the accumulate weight.
- 2) **Determining the risk profile.** To determine the risk profile for the application (code percentages in each risk category), we take the maximum accumulated weight where the value of the SCM is equal or less than the risk threshold for the current risk in the profile.

A.2.3. Deriving Ratings of Applications

The calculation of the rating of an application is based on all ratings of all SCMs of the application. Figure A.6 shows the overall process for deriving rating of applications. First, are derived the ratings for the SCMs of the application based on the risk profiles of the SCMs and in a set of rating thresholds for each SCMs. Then, these ratings are combined and averaged for deriving the rating of the application (i.e., 1 to 5 stars). Next sections details the stages of the process.

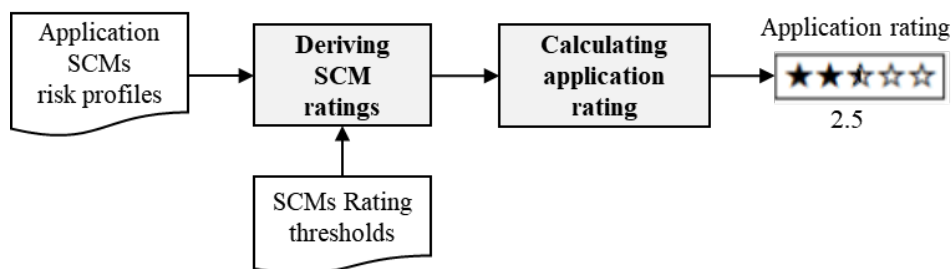


Figure A.6.: Process for deriving ratings of applications.

A.2.3.1. Deriving Ratings of the SCMs of Applications

The percentages of code in the risk profile are compared against the rating thresholds for determining the rating of the SCM. From this process results a set of ratings for each application.

Figure A.7 includes a table with a set of rating thresholds for mapping risk profiles into ratings (1 to 5 stars). The risk thresholds are defined in the table headers, and the rating thresholds are defined in the table body. Each set of rating thresholds defines the cumulative upper-boundaries for the moderate, high and very-high risk categories.

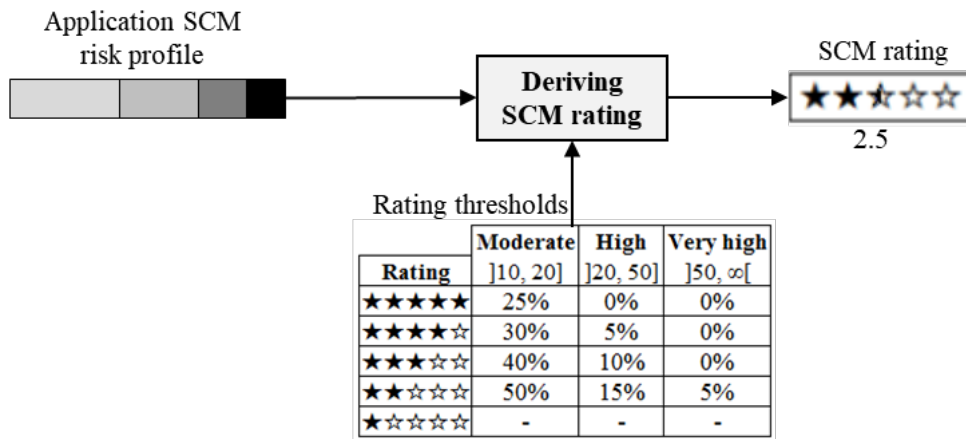


Figure A.7.: Process for deriving SCM ratings.

To derive the rating for a risk profiles, we chose the lowest index of the line of the table’s rating thresholds where the risk profile fits. For example, a risk profile with the values 29% for Moderate (fits in lines 2-5), 4% for High (fits in lines 2-5) and 0% for Very high (fits in line 1-5), has a risk rating profile of four stars (line 2 of the column *Rating* in the table), because it is the minimum line where all the percentages of code fit.

The ratings of the SCM will need to be further combined and aggregated and for that, a discrete scale is not adequate. For instance, percentages of code: 30, 5, 0 have a rating with 4 stars and percentages of code: 40, 10, 0 have rating of 3 stars. However, percentages of code between the limits (i.e., 31-39, 5-9, 0-0) also have 3 stars. The use of a continuous scale has the advantage of providing more precise results. With this approach, percentages of code in the middle of the limits of the previous examples (35, 5, 0) have 3.5 stars.

An equivalence of the two scales can be established using a linear interpolation based on the limit values. The discrete scale of the rating can be converted to a continuous interval in [0.5, 5.5] limits, using a linear interpolation function per risk category and choosing the minimum value as the rating. This is obtained by starting to determinate the discrete rating (*DR*) using a discrete scale. Then, we use the formula: $\text{continuous rating} = DR + 0.5 - (p - t_0) / (t_1 - t_0)$. Where p is the percentage of volume in the risk profile, and t_0, t_1 are the lower and upper limits for the rate *DR*. As an example, for the risk profile with the percentages, 36% for moderate risk, 6% for high risk, and 0% for very high risk, and based on the rating threshold in the Figure A.7, the *DR* is 3. Thus, the ratings per risk category for a continuous scale are:

- moderate risk – $3 + 0.5 - (36 - 30)/(40.0 - 30.0) = 2.9$;
- high risk – $3 + 0.5 - (6.0 - 5.0)/(10.0 - 5.0) = 3.3$;
- very-high risk – $3 + 0.5 = 3.5$.

At the end, the final continuous rating is 2.9 because it is the minimum value of them all.

A.2.3.2. Calculating the Application Rating

The step for calculating the application rating is an easy one. All [SCMs](#) ratings are averaged to calculate the overall application rating. For a discrete scale, the ratings are rounded up.

A.2.3.3. Calculating the Rating Thresholds

As said before, we need one table of rating thresholds for each SCM. Alves et al. [223] proposed an algorithm for obtaining the rating thresholds based on the risk profiles of a large set of applications and a set of risk thresholds. In fact, the algorithm computes the rating thresholds, which allows to split the set of applications according to a desired distribution of the applications per rating. Therefore, for a normal distribution of 5-20-20-20-5, the algorithm calculates the rating thresholds to split the code into five ratings. As an example, by applying these rating thresholds we can get to know what are the best 5% of applications in the set of applications. The method is applied by SIG to annually re-calibrate the SIG quality model [226], which forms the basis of the evaluation and certification of software maintainability conducted by SIG [188] and TÜViT [181].

In this work, we use the mean of the lowest and the highest rating thresholds resulting from a stability analysis, performed by Alves et al. [223], of the variability of the rating thresholds for 100 runs, randomly sampling 90% of the applications in the set of applications (100). The resulting rating tables are given in Tables [A.3-A.10](#).

A.3. Rating Thresholds Tables

Table A.3.: Rating Thresholds for Duplication

Rating	Duplication
★★★★★	3.0
★★★★☆	5.0
★★★☆☆	10.0
★★☆☆☆	20.0
★☆☆☆☆	-

Table A.4.: Unit Size (LLOC)

	Moderate	High	Very high
Rating]13 ,21]]21 ,39]]39 , +∞[
★★★★★	19.6	10.0	3.8
★★★★☆	26.4	16.2	6.8
★★★☆☆	31.2	22.8	11.2
★★☆☆☆	38.4	29.4	15.8
★☆☆☆☆	-	-	-

Table A.5.: Unit Complexity (CCN2)

	Moderate	High	Very high
Rating]18 ,29]]29 ,63]]63 , +∞[
★★★★★	19.6	10.0	3.8
★★★★☆	26.4	16.2	6.8
★★★☆☆	34.7	23.6	11.4
★★☆☆☆	44.8	16.7	18.4
★☆☆☆☆	-	-	-

Table A.6.: Coupling Between Objects (CBO)

	Moderate	High	Very high
Rating]1 ,2]]2 ,3]]3 , +∞[
★★★★★	24.5	14.0	6.6
★★★★☆	30.5	19.7	9.2
★★★☆☆	35.7	22.8	11.7
★★☆☆☆	45.1	32.7	17.9
★☆☆☆☆	-	-	-

Table A.7.: Unit testing (NPATH)

	Moderate	High	Very high
Rating]256 ,5612]]5612 ,181193]]181193 , +∞[
★★★★★	24.5	14.0	6.6
★★★★☆	30.5	19.7	9.2
★★★☆☆	35.7	22.8	11.7
★★☆☆☆	45.1	32.7	17.9
★☆☆☆☆	-	-	-

Table A.8.: Size – Weighted Class Methods (WCM)

	Moderate	High	Very high
Rating]117 ,176]]176 ,321]]321 , +∞[
★★★★★	19.6	10.0	3.8
★★★★☆	26.4	16.2	6.8
★★★☆☆	34.7	23.6	11.4
★★☆☆☆	44.8	16.7	18.4
★☆☆☆☆	-	-	-

Table A.9.: Interfacing - Number of parameters per unit (NPARM)

	Moderate	High	Very high
Rating]1 ,2]]2 ,3]]3 , +∞[
★★★★★	12.1	5.2	2.2
★★★★☆	15.3	7.3	3.2
★★★☆☆	19.2	9.3	4.8
★★☆☆☆	50.0	15.4	6.6
★☆☆☆☆	-	-	-

Table A.10.: Class Interface Size - Number of non-private methods and properties

	Moderate	High	Very high
Rating]23 ,34]]34 ,48]]48 , +∞[
★★★★★	12.1	5.2	2.2
★★★★☆	15.3	7.3	3.2
★★★☆☆	19.2	9.3	4.8
★★☆☆☆	50.0	15.4	6.6
★☆☆☆☆	-	-	-

List of WordPress plugins

Tables B.1 to B.4 lists the WordPress plugins, including rating information and vulnerabilities as result of the process of [Collecting the Source Code of Vulnerable Applications](#) (Section 4.2.1) of the benchmark instantiation in Section 4.2. The tables contains data as follows:

- Table B.1: Highest-quality.
- Table B.2: High-quality.
- Table B.3: Medium-quality.
- Table B.4: Low-quality.

Table B.1.: List of WordPress plugins for Highest-quality scenario.

Plugin	Code Type	Rating	Analysability	Changeability	Stability	Testability	TP_SQLi	FP_SQLi	TP_XSS	FP_XSS	WPVD_SQLi	WPVD_XSS
1 advertizer.1.0	OOP	5	5	5	5	5	1	0	2	0	1	0
2 calculated-fields-form.1.0.10	POP	4.75	4	5	5	5	5	0	8	0	3	0
3 contact-form.2.7.5	OOP	4.5	4	5	4	5	5	0	20	1	1	0
4 facebook-opengraph-meta-plugin.1.0	OOP	4.75	5	5	5	4	3	0	18	2	1	0
5 faqs-manager.1.0	OOP	4.5	4	5	4	5	7	0	1	0	1	0
6 landing-pages.1.8.7	OOP	4.5	5	5	4	4	1	1	47	6	1	1
7 occasions.1.0.4	OOP	4.5	3	5	5	5	0	0	0	0	0	0
8 photoracer	OOP	4.5	5	5	4	4	5	0	29	1	4	2
9 simple-support-ticket-system.1.2	POP	5.25	5	5	5.5	5.5	22	0	8	3	5	0
10 simply-poll.1.4.1	OOP	4.75	4	5	5	5	1	0	8	12	0	0
11 stripshow.2.5.2	OOP	4.5	4	5	4	5	3	0	8	18	0	0
12 wpforum.1.7.8	OOP	4.5	3	5	5	5	22	4	19	2	0	0

Table B.2.: List of WordPress plugins for High-quality scenario.

Plugin	Code Type	Rating	Analysability	Changeability	Stability	Testability	TP_SQLi	FP_SQLi	TP_XSS	FP_XSS	WPVD_SQLi	WPVD_XSS
1 ajaxgallery.3.0	POP	3.5	2	3	4	5	3	0	2	0	1	0
2 all-video-gallery.1.2	OOP	4.25	4	5	4	4	9	0	80	0	0	0
3 bigcontact.1.4.6	OOP	3.75	4	4	2	5	3	0	0	0	3	0
4 collision-testimonials.3.0	OOP	3.5	2	4	5	3	11	3	19	0	2	0
5 contus-hd-flv-player.1.3	OOP	3.75	3	4	5	3	9	0	60	1	1	0
6 contus-video-gallery.2.8	OOP	3.5	4	4	4	2	12	0	21	5	1	0
7 couponer.1.2	OOP	4.38	3	5	5.5	4	2	0	29	0	1	0
8 cp-multi-view-calendar.1.1.7	OOP	3.5	3	4	4	3	1	0	26	0	0	0
9 cp-reservation-calendar.1.1.6	POP	4	3	3	5	5	5	0	14	2	1	0
10 disqus-comment-system.2.02.2812	OOP	3.75	3	4	4	4	0	0	4	0	0	0
11 dynamic-font-replacement-4wp.1.3	POP	3.88	2	3	5.5	5	5	0	11	2	1	0
12 easy-career-openings	OOP	4.38	3	4	5.5	5	5	0	21	0	0	0
13 events-registration.5.44	POP	3.5	3	3	5	3	55	0	415	7	0	0
14 external-video-for-everybody.2.1.1	POP	4	3	3	5	5	0	0	0	0	0	0
15 feedweb.3.0.6	POP	4.25	3	4	5	5	0	0	10	4	0	0
16 forum-server.1.7.1	OOP	3.75	4	4	4	3	12	4	4	0	0	0
17 freshmail-newsletter.1.5.8	OOP	3.75	5	4	3	3	1	0	35	5	1	0
18 fs-real-estate-plugin.2.06.01	OOP	3.75	3	4	4	4	48	9	82	0	4	0
19 gallery-bank.3.0.101	POP	4.25	4	5	3	5	0	1	12	31	0	0
20 gallery-images.1.0.1	OOP	3.5	3	4	4	3	13	4	8	0	1	0
21 gallery-objects.0.4	POP	3.5	3	3	5	3	0	0	24	17	0	0
22 gb-gallery-slideshowtags1.5	OOP	4.25	4	5	5	3	2	0	9	0	1	0
23 global-content-blocks.1.2	OOP	3.75	2	4	4	5	4	0	11	0	1	0
24 google-document-embedder.2.5.16	POP	3.75	3	3	4	5	1	0	10	1	1	0
25 ip-blacklist-cloud.3.4	POP	3.75	4	3	5	3	11	5	139	2	0	0
26 jaspreetchahals-coupons-lite.2.8	OOP	3.75	2	4	5	4	0	0	29	0	0	0
27 media-library-categories.1.0.6	OOP	3.5	3	4	2	5	3	0	20	2	1	0
28 mtouch-quiz.3.06	POP	3.75	4	2	5	4	6	0	102	2	0	0
29 mystat.2.6	OOP	3.75	3	5	4	3	4	0	23	2	1	0
30 mz-jajak.2.1	POP	3.63	3	3	5.5	3	9	1	41	4	2	0
31 newstatpress.1.0.4	POP	3.75	2	4	5	4	2	0	81	4	0	0
32 nextgen-smooth-gallery.1.2	OOP	3.75	2	4	5	4	0	0	2	1	0	0
33 oqey-gallery.0.4.8	OOP	3.75	3	5	2	5	1	1	68	4	1	0
34 oqey-headers.0.3	POP	4.13	3	3	5	5.5	3	0	2	0	1	0
35 post-highlights.2.2	OOP	4	3	4	4	5	2	0	7	3	1	0
36 proplayer.4.7.9.1	OOP	4	4	4	3	5	0	0	3	0	0	0
37 pure-html.1.0.0	POP	3.5	2	4	4	4	1	0	7	0	1	0
38 related-sites.2.1	OOP	3.75	3	4	4	4	1	0	0	0	1	0
39 scormcloud.1.0.6.6	OOP	3.5	4	3	2	5	10	0	31	12	1	0
40 sendit.2.1.0	OOP	4	4	5	4	3	19	0	58	2	0	0
41 sh-slideshow.3.1.4	OOP	4.25	3	5	5	4	5	0	27	0	1	0
42 simple-forum.4.3.0	OOP	3.75	3	5	3	4	16	33	153	73	0	0
43 simple-login-log.0.9.3	OOP	3.5	3	4	2	5	0	0	0	1	0	0
44 slider-image.2.6.8	OOP	3.5	3	4	4	3	22	1	14	0	0	0
45 spider-facebook.1.0.8	OOP	3.5	3	4	4	3	0	0	19	0	0	0
46 timeline.0.1	OOP	4	3	4	5	4	0	0	9	0	0	0
47 trafficanalyzer.3.4.2	OOP	3.75	4	4	3	4	3	0	9	2	0	0
48 videos-html5-video-player-for-wordpress.4.5.0	POP	3.75	3	3	4	5	0	0	1	0	0	0
49 videowhisper-video-presentation.1.1	OOP	4	2	5	5	4	4	0	7	8	1	0
50 wp125.1.5.3	OOP	4	3	4	5	4	1	0	14	3	0	0
51 wp-ds-faq.1.3.2	OOP	3.75	2	5	4	4	8	0	29	10	1	0
52 wp-powerplaygallery.3.3	OOP	3.5	4	3	4	3	2	0	11	5	1	0
53 wp-predict.1.0	OOP	3.5	3	4	4	3	8	0	0	2	0	0
54 wp-statistics.9.4	OOP	3.75	4	4	2	5	3	0	11	7	0	1
55 wp-survey-and-poll.1.1	OOP	3.5	2	4	5	3	1	0	1	0	1	0
56 yawpp.1.2	POP	4.25	3	4	5	5	0	0	17	0	0	0

Table B.3.: List of WordPress plugins for Medium-quality scenario.

Plugin	Code Type	Rating	Analysability	Changeability	Stability	Testability	TP_SQLi	FP_SQLi	TP_XSS	FP_XSS	WPVD_SQLi	WPVD_XSS
1 adrotate.3.9.4	OOP	3.25	3	4	4	2	0	0	74	24	0	0
2 another-wordpress-classifieds-plugin.2.2.1	OOP	2.5	2	3	3	2	3	0	10	11	0	0
3 calendar.1.3.3	POP	3.25	3	3	4	3	0	0	23	1	0	0
4 content-audit.1.6	POP	3.25	2	3	4	4	1	0	1	0	1	0
5 contextual-related-posts.2.0.1	OOP	2.75	2	2	5	2	0	0	2	1	0	0
6 copyright-licensing-tools.1.4	POP	2.75	3	2	3	3	1	0	8	0	1	0
7 count-per-day.3.4	OOP	3.25	4	3	3	3	1	0	35	15	1	0
8 crawlrate-tracker.2.02	OOP	3.25	4	3	3	3	1	0	14	1	1	0
9 dukapress.2.5.9	OOP	2.5	3	3	2	2	2	0	62	22	1	0
10 duplicator.0.5.14	OOP	3	3	3	2	4	14	10	33	17	1	1
11 editorial-calendar.2.6	OOP	2.5	2	3	3	2	0	0	5	1	0	0
12 eventify.1.7.f	OOP	2.5	3	3	3	1	6	0	12	0	1	0
13 events-manager.5.5.3.1	OOP	2.5	4	2	2	2	0	0	33	77	0	0
14 fbpromotions.1.3.3	OOP	3.25	5	3	1	4	1	0	147	23	1	0
15 flash-album-gallery.2.55	OOP	3	4	3	1	4	5	10	195	58	0	0
16 ip-logger.3.0	OOP	3.25	4	2	3	4	13	0	14	0	1	0
17 js-appointment.1.5	OOP	2.75	4	3	2	2	85	0	287	3	1	0
18 knr-author-list-widget.2.0.0	OOP	3.25	3	4	4	2	1	0	38	3	1	0
19 leaguemanager.3.8	OOP	3	4	3	1	4	0	0	61	10	0	0
20 levelfourstorefront.8.1.14	OOP	2.75	4	3	1	3	42	103	511	9	0	0
21 login-with-ajax.3.1.4	OOP	2.75	3	3	2	3	0	0	3	0	0	0
22 mail-subscribe-list.2.1.1	POP	3	2	3	3	4	0	1	1	0	0	0
23 mingle-forum.1.0.33.3	OOP	3	4	2	2	4	9	13	22	12	0	0
24 montezuma.1.1.7	OOP	2.75	3	3	3	2	0	0	5	0	0	0
25 my-category-order.2.8	POP	3.38	1	3	5.5	4	1	0	8	0	1	0
26 newsletter.3.6.4	OOP	2.5	4	2	1	3	2	0	19	36	0	1
27 odihost-newsletter-plugin	OOP	2.5	3	2	1	4	18	1	21	0	1	0
28 paid-downloads.2.01	OOP	3	3	3	4	2	1	5	4	3	1	0
29 paypal-digital-goods-monetization-powered-by-cleeng.2.2.16	OOP	3	3	3	2	4	2	0	3	0	0	0
30 photo-gallery.1.2.8	OOP	3	4	3	3	2	0	0	7	222	0	0
31 profiles.2.0.RC1	POP	3.25	2	3	3	5	9	9	33	0	1	0
32 qtranslate.2.5.39	OOP	3.25	2	3	4	4	1	0	43	3	0	0
33 relevanssi.3.2	POP	2.5	2	2	3	3	0	0	0	0	0	0
34 social-media-widget.4.0.2	OOP	3	3	3	5	1	0	0	0	0	0	0
35 sp-client-document-manager.2.4.1	OOP	2.5	3	2	3	2	1	0	95	2	1	0
36 store-locator.3.33.1	OOP	3.25	2	4	4	3	0	2	64	18	0	0
37 taggator.1.54	POP	3.25	1	3	5	4	0	0	2	0	0	0
38 tune-library.1.5.2	OOP	3.25	2	3	5	3	0	0	0	2	0	0
39 tweet-old-post.3.2.5	OOP	3.25	2	3	4	4	1	0	11	1	1	0
40 upm-polls.1.0.4	OOP	3.25	3	4	2	4	0	0	30	3	0	0
41 visual-form-builder.2.8.2	OOP	3	4	3	2	3	2	0	74	5	2	0
42 woocommerce.2.3.0	OOP	3	4	3	2	3	2	1	6	47	1	2
43 wordpress-seo.1.7.3.3	OOP	2.5	3	2	1	4	0	0	2	10	0	0
44 wp-audio-gallery-playlist.0.12	POP	3.38	1	3	5.5	4	1	0	4	0	1	0
45 wp-championship.5.8	OOP	3	3	3	5	1	18	5	6	3	0	0
46 wp-imagezoom.1.0.0	POP	3.25	2	3	3	5	0	0	3	0	0	0
47 wp-menu-creator.1.1.7	OOP	2.75	4	3	3	1	13	0	6	8	1	0
48 wp-photo-album-plus.5.4.18	OOP	3.25	3	4	4	2	0	1	135	28	0	0
49 wp-realty.1.0.1	POP	2.75	2	1	5	3	0	0	0	0	0	0
50 wp-spamfree.3.2.1	OOP	3	2	3	5	2	0	0	0	0	0	0
51 wp-symposium.14.10	OOP	2.5	4	2	2	2	10	2	217	1	0	0

Table B.4.: List of WordPress plugins for Low-quality scenario.

Plugin	Code Type	Rating	Analysability	Changeability	Stability	Testability	TP_SQLi	FP_SQLi	TP_XSS	FP_XSS	WPVD_SQLi	WPVD_XSS
1 community-events.1.2.9	OOP	2	2	3	2	1	3	0	18	0	0	0
2 easy2map.1.2.4	OOP	2.25	4	1	2	2	8	0	9	8	4	2
3 evarisk.5.1.3.6	OOP	1.75	3	2	1	1	16	0	92	45	1	0
4 events-calendar.6.6	OOP	2.25	2	2	3	2	0	0	11	0	0	0
5 feedwordpress.2015.0426	OOP	2.25	3	2	1	3	1	0	0	0	1	0
6 funcaptcha.1.2.1	OOP	2.25	3	1	3	2	0	0	4	2	0	0
7 gigpress.2.3.8	OOP	1.25	2	1	1	1	2	0	105	5	2	1
8 global-flash-galleries.0.15.3	OOP	2.25	2	2	2	3	0	0	13	6	0	0
9 pie-register.2.0.18	OOP	2	3	2	2	1	5	1	42	3	2	2
10 pods.2.4.4	OOP	1.75	3	2	1	1	1	0	13	17	0	0
11 quartz.1.01.1	POP	2.25	2	1	3	3	3	0	7	1	0	0
12 sermon-browser.0.43	OOP	2	3	2	2	1	8	31	101	5	0	0
13 syntaxhighlighter.3.1.10	OOP	1.25	2	1	1	1	0	0	0	0	0	0
14 usc-e-shop.1.3.12	OOP	2.25	4	2	1	2	3	0	130	25	0	0
15 xili-language.2.15.2	OOP	2.25	3	2	1	3	0	0	0	0	0	0

Benchmarking Procedure and Rules

This Appendix provides details about the *Benchmarking Procedure* of the benchmark instantiation presented in Section 4.3, which instantiates the *Benchmark Approach* proposed (see Section 4.1) in this thesis.

The organization of the appendix is as follows. Section C.1 presents the preparation. Section C.2 details the execution. Section C.3 details the normalization of reports of the outputs of the SAST tools. Section C.4 details the process vulnerability verification. Section C.5 presents metrics calculation and ranking of the SAST tools.

C.1. Preparation

In the stage *Preparation* (Section 4.1.4) of the benchmark run, we have to select the SAST tools to be benchmarked and to specify the configuration settings for the tools.

We focus on free SAST tools as both occasional developers and professional software houses wanting to speed up the development process and reduce cost tend to use free tools as much as possible. Furthermore, such tools are easily available for research and results can be published without infringing licensing agreements. In practice, we evaluated the following tools: RIPS v0.55 [51], Pixy v3.03 [81], phpSAFE [109], WAP v2.0.1 [103], and WeVerca v20150804 [100]. RIPS and Pixy are the two most referenced PHP SAST tools in the literature, but they are not ready for OOP analysis. Pixy performs tainted analysis and alias analysis, but has not been updated since 2007, and RIPS has only been developed as open source until 2014. Then, “RIPS Technologies GmbH¹” released a commercial version of RIPS able to fully analyze OOP code [196]. WAP,

¹<https://www.ripstech.com>

phpSAFE, and WeVerca are more recent tools under active development, and they are prepared for OOP code. In terms of configuration, phpSAFE, RIPS, WAP and Pixy are configured by default for PHP EPs, SSs and sanitization functions (e.g., `htmlentities`, `mysql_real_escape_string`). WeVerca does not allow configuration and includes, out of the box, a programmed list of EPs, SSs and SFs.

Usually, projects have a lot of source files organized in several folder and sub-folders. Except for RIPS, all the other tools were configured to search vulnerabilities in a single file at each run because it can increase the rate of source code files successfully analyzed by the tools. In fact, SAST tools can crash or can block while analyzing a file. For instance, a tool analyzing a list of source code files can crash in the middle of the list, leaving to analyze the remaining half.

To automate the process of running the SAST tools, we developed shells scripts, based on the list of all source code files in the workload. These scripts are able to run the tools, analyzing one file per run, and storing the results in a separate file.

The configuration settings of the SAST tools under benchmarking is detailed next:

- 1) **phpSAFE**: was used to characterize the VLOCs and the NVLOCs in the workload. For instance, we used the same configuration described previously in Section 4.2.3.
- 2) **RIPS**: like for phpSAFE we used the same consideration as described in Section 4.2.3.
- 3) **Pixy**: is a command line tool and provides a text-based report of the vulnerabilities offering several verbosity levels. A key limitation is the absence of capabilities to analyze OOP code. Pixy analyses one file a time and does not analyses the included files in the file being analyzed. Pixy was configured with the following options: `-y xss:sql` for searching XSS and SQLi vulnerabilities; `-f` to print function information; `-A` to use alias analysis to cope with the PHP reference operator `&` (ex. `$variable1=& $variable2`); `-g` to disable `register_globals` for analysis; `-o` to set the output directory (for graphs etc.) to store all the results as a set of log files.
- 4) **WAP**: is a command line source code static analysis and data mining tool to detect and correct input validation vulnerabilities in web applications written in PHP. WAP was configured with the following options: `-a` to detect vulnerabilities, without corrected them; `-sqli` to detect SQLi vulnerabilities; `-xss` to detects XSS vulnerabilities; `-out` to forwards the `stdout` to a specified file to store the results. Despite the ability of the tool to analyze a full project at once, we used the option to analyze one source file per run by providing the file name. The results of each run are stored in a separate file.
- 5) **WeVerca**: is a static analysis framework for web applications written in PHP. The aim of the framework is to allow easy specification of precise static analyses. The framework has been used to develop a tool for securing web applications by reporting suspicious code constructs and commands. It detects XSS and SQLi vulnerabilities using static taint analysis. WeVerca was developed as a library, however it also contains a command line able to analyze a list of files containing PHP code. WeVerca was configured with the following options: `-sa` to perform static analysis; and the `>` shell symbol to redirect the console output to a file.

C.2. Execution

We ran the benchmark for all the [SAST](#) tools searching for [XSS](#) and [SQLi](#) vulnerabilities in the workload made from the plugins. Overall, [WAP](#) was able to analyze all plugins, but seven of them only partially. [Pixy](#) analyzed partially 103 plugins (i.e., fails in 1,473 files) and [WeVerca](#) was not able to analyze 20 source files of 14 plugins. [phpSAFE](#) was unable to fully analyze 18 plugins (130 files), taking a very long time on those plugins without returning any results. [RIPS](#) outputted the message “*Code is object-oriented. This is not supported yet and can lead to false negatives*” for 76 plugins (2,179 files). In practice, the tools could not fully analyze some plugin/files, reporting runtime errors or taking a very long time without any results. This results from limitations of the static analysis tools used, potentially due to the size/complexity of some files.

C.3. Normalization of Reports

The vulnerability [reports](#) of the [SAST](#) tools have specific formats. Therefore, we normalized the reports in a common format to characterize the [location](#) of the vulnerabilities with the data fields listed in [Table C.1](#). Thus, we developed a script per tool to convert its results to the common format.

Table C.1.: Common format for the results of [SAST](#) tools.

Field	Description
Application	Name of the plugin
Vulnerability type	Examples are: XSS and SQLi
Source code file name	The file name includes the path name inside of the plugin directory structure
Number of LOC	The number of the source code line with the vulnerable SS
Entry Points (EPs)	The list of the EPs on which the vulnerable variable(s) depend (e.g., <code>GET PHP array</code>)
Vulnerable Variables	The variables used as parameter in the SSs
Sensitive Sinks (SSs)	The name of the vulnerable function (e.g., <code>print</code> , <code>mysql_query</code>)

C.4. Vulnerability Verification

The reports of the [SAST](#) tools are in the same format of the list of [VLOCs](#) and [NVLOCs](#) of the workload. Therefore, the correctness of the vulnerabilities reported by the tools was automatically verified by a grading program developed to perform this task. Unfortunately, the tools can report vulnerabilities that belong to the set of [SSs](#) of the list [NVLOCs](#) that require a manual verification to confirm their veracity, and then update the lists of [VLOCs](#) and [NVLOCs](#) according to the results of the manual review:

- 1) **if the vulnerability reported is a [TP](#)**. To remove the entry from the set of [SS](#) of the list of [NVLOCs](#) and to add this entry do the list of [VLOCs](#).
- 2) **if the vulnerability reported is a [FP](#)**: in the list of [NVLOCs](#) move the entry from the set of [SS](#) to the set of [FP](#).

Table C.2.: Evolution of P and N instances with the number of SAST tools.

Scenario	Initial (I)		WAP		Pixy		WeVerca		Final (F)		F-I		
	P	N	P_i	N_i	P_i	N_i	P_i	N_i	P	N	P	N	
SQLi	1	41	89	34	-2	0	0	0	0	75	87	34	-2
	2	308	1,128	31	-7	0	0	7	-6	346	1,115	38	-13
	3	251	2,216	15	-1	0	0	1	-1	267	2,214	16	-2
	4	46	1,137	4	0	0	0	0	0	50	1,137	4	0
Total	646	4,570	84	-10	0	0	8	-7	738	4,553	92	-17	
XSS	1	168	990	0	0	0	0	0	0	168	990	0	0
	2	1,767	5,896	4	-3	0	0	71	-68	1,842	5,825	75	-71
	3	2,315	10,022	71	-50	0	0	3	-3	2,389	9,969	74	-53
	4	535	3,597	9	-2	0	0	1	-1	545	3,594	10	-3
Total	4,785	20,505	84	-55	0	0	75	-72	4,944	20,378	159	-127	
Total	5,431	25,075	168	-65	0	0	83	-79	5,682	24,931	251	-144	

To have an idea of the additional amount of manual review required, we proceeded with the *vulnerability verification* as follows. First, we begin with the number of P and N resulting from the process of *Identifying Vulnerabilities and Non-vulnerabilities* (Section 4.2.3) using the tools phpSAFE and RIPS, and then we successively updated Positive Instances (P) and Negative Instances (N) as result of adding the results of one SAST tool at a time.

Table C.2 shows the evolution of the P and N as result of adding one tool to the list of tools used to create the workload (RIPS and phpSAFE). The columns P_i and N_i represent the increment of P and N as result of adding the respective tool. As shown, adding the first tool (WAP) it adds 168 new VLOCs and moves 65 (10+55) NVLOCs from the list of NVLOCs to the list of VLOCs. This occurred because the tool reports vulnerabilities previously unknown included in the list of NVLOCs that was created based on the identification of SSs (i.e., not manually reviewed). The values of P and N do not change when adding the Pixy tool. Like WAP, WeVerca adds several vulnerabilities (83) and moves 79 NVLOC to the list of VLOC. Running this benchmark instantiation required 251 (168 + 83) manual reviews. It is important to emphasize that the sum of the of number of P and N instances can not change.

The ranking of the SAST tools is relative to the actual number of P and N instances. In fact, there are large number of SSs not reviewed. From the initial total number of N instances (25075, table 4.8), as result of using the SAST tool RIPS and phpSAFE were reviewed only 5.75% (i.e., (534 + 908) / 25075) of the SSs. To obtain absolute rankings requires to review all the SSs not manually reviewed (i.e., 23633). This is difficult and time consuming task but, it allows to gain ground truth of the workload.

C.5. Metrics Calculation and Ranking

Based on the SAST tools outputs and their verification (previous step, Section C.4) and based on the number of P and N instances by scenario and class of vulnerability shown in Table C.2, the benchmark metrics are calculated automatically. Afterwards, SAST tools are ranked according to the metrics recommend for each scenario (see Table 4.1). The results are listed in Table 4.9

and Table 4.10).

APPENDIX D

Results for all Combinations of five SAST Tools: WordPress Plugins

Tables [D.1](#) to [D.5](#) lists all results of combining the results of five [SAST](#) tools analyzing WordPress plugins using *1-out-of-n* strategy (see [Section 5.1](#) for more details). The tables contains data results as follows:

- [Table D.1](#): Highest-quality.
- [Table D.2](#): High-quality.
- [Table D.3](#): Medium-quality.
- [Table D.4](#): Low-quality.
- [Table D.5](#): Regardless scenarios.

Table D.1.: Best Solutions for the WordPress plugins: SQLi, XSS and SQLi + XSS: Highest-quality

SQLi							XSS							SQLi + XSS						
Tools	TP	FP	Pg	MM	TM	P/R	Tools	TP	FP	Pg	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R	
Highest-quality			Rec.	Prec.					Rec.	Prec.					Rec.	Prec.				
ac	65	5	9	0.867	0.929	-	ab	165	43	11	0.982	0.793	-	abc	230	50	0.947	0.821	-	
ace	65	5	9	0.867	0.929	-	abe	165	43	11	0.982	0.793	-	abcd	230	50	0.947	0.821	-	
abce	65	5	9	0.867	0.929	-	abc	165	45	11	0.982	0.786	-	abcde	230	50	0.947	0.821	-	
acde	65	5	9	0.867	0.929	-	abd	165	45	11	0.982	0.786	-	abce	230	50	0.947	0.821	-	
abc	65	5	9	0.867	0.929	-	abce	165	45	11	0.982	0.786	-	acde	205	31	0.844	0.869	-	
acd	65	5	9	0.867	0.929	-	abde	165	45	11	0.982	0.786	-	acd	204	29	0.840	0.876	-	
abcd	65	5	9	0.867	0.929	-	abcde	165	45	11	0.982	0.786	-	ab	194	48	0.798	0.802	-	
abcde	65	5	9	0.867	0.929	-	abcd	165	45	11	0.982	0.786	-	abe	194	48	0.798	0.802	-	
ce	49	4	7	0.653	0.925	-	ade	140	26	11	0.833	0.843	-	abd	194	50	0.798	0.795	-	
c	49	4	7	0.653	0.925	-	acde	140	26	11	0.833	0.843	-	abde	194	50	0.798	0.795	-	
bc	49	4	7	0.653	0.925	-	ad	139	24	10	0.827	0.853	-	ace	184	27	0.757	0.872	-	
cd	49	4	7	0.653	0.925	-	acd	139	24	10	0.827	0.853	-	bcde	180	41	0.741	0.814	-	
bce	49	4	7	0.653	0.925	-	bcde	131	37	11	0.780	0.780	-	bce	179	38	0.737	0.825	-	
cde	49	4	7	0.653	0.925	-	bce	130	34	11	0.774	0.793	-	bcd	173	41	0.712	0.808	-	
bcde	49	4	7	0.653	0.925	-	bde	128	35	11	0.762	0.785	-	ac	172	25	0.708	0.873	-	
bcd	49	4	7	0.653	0.925	-	be	126	30	11	0.750	0.808	-	ade	169	31	0.695	0.845	-	
a	29	5	5	0.387	0.853	-	bcd	124	37	11	0.738	0.770	-	ad	168	29	0.691	0.853	-	
ae	29	5	5	0.387	0.853	-	bd	121	35	11	0.720	0.776	-	bc	166	37	0.683	0.818	-	
ab	29	5	5	0.387	0.853	-	ace	119	22	11	0.708	0.844	-	ae	143	25	0.588	0.851	-	
ad	29	5	5	0.387	0.853	-	bc	117	33	10	0.696	0.780	-	cde	133	22	0.547	0.858	-	
abe	29	5	5	0.387	0.853	-	ae	114	20	11	0.679	0.851	-	a	131	23	0.539	0.851	-	
ade	29	5	5	0.387	0.853	-	b	113	29	10	0.673	0.796	-	bde	128	35	0.527	0.785	-	
abde	29	5	5	0.387	0.853	-	ac	107	20	9	0.637	0.843	-	be	126	30	0.519	0.808	-	
abd	29	5	5	0.387	0.853	-	a	102	18	8	0.607	0.850	-	cd	124	20	0.510	0.861	-	
e	0	0	0	0.000	-	-	cde	84	18	8	0.500	0.824	-	bd	121	35	0.498	0.776	-	
be	0	0	0	0.000	-	-	de	78	16	8	0.464	0.830	-	b	113	29	0.465	0.796	-	
d	0	0	0	0.000	-	-	cd	75	16	7	0.446	0.824	-	ce	106	13	0.436	0.891	-	
de	0	0	0	0.000	-	-	d	69	14	7	0.411	0.831	-	de	78	16	0.321	0.830	-	
b	0	0	0	0.000	-	-	ce	57	9	8	0.339	0.864	-	c	72	10	0.296	0.878	-	
bde	0	0	0	0.000	-	-	e	44	5	7	0.262	0.898	-	d	69	14	0.284	0.831	-	
bd	0	0	0	0.000	-	-	c	23	6	3	0.137	0.793	-	e	44	5	0.181	0.898	-	

MM - Main Metric. TM - Tiebreaker Metric. Rec. - Recall. Prec. - Precision. Pg - # of plugins

Infor. - Informedness. F-Meas. - F-Measure. Marked. - Markedness.

Tools: a - phpSAFE. b - RIPS. c - WAP. d - Pixy. e - WeVerca.

Table D.2.: Best Solutions for the WordPress plugins: SQLi, XSS and SQLi + XSS: High-quality

SQLi							XSS						SQLi + XSS							
Tools	TP	FP	Pg	MM	TM	P/R	Tools	TP	FP	Pg	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R	
High-quality			Infor.	Rec.	Prec.		Infor.			Rec.	Prec.		Infor.			Rec.	Prec.			
acde	318	59	36	0.866	0.919	0.844	abce	1841	224	51	0.961	1.000	0.892	abcde	2159	284	0.946	0.987	0.884	
abce	318	60	36	0.865	0.919	0.841	abcde	1841	224	51	0.961	1.000	0.892	abce	2159	284	0.946	0.987	0.884	
abcde	318	60	36	0.865	0.919	0.841	abe	1838	223	51	0.960	0.998	0.892	abde	2124	283	0.930	0.971	0.882	
ace	316	59	36	0.860	0.913	0.843	abde	1838	223	51	0.960	0.998	0.892	abe	2124	283	0.930	0.971	0.882	
acd	311	58	35	0.847	0.899	0.843	abc	1770	224	51	0.922	0.961	0.888	abc	2081	284	0.910	0.951	0.880	
abc	311	60	35	0.845	0.899	0.838	abcd	1770	224	51	0.922	0.961	0.888	abcd	2081	284	0.910	0.951	0.880	
abcd	311	60	35	0.845	0.899	0.838	abd	1767	223	51	0.921	0.959	0.888	abd	2046	283	0.894	0.935	0.878	
ac	306	58	35	0.832	0.884	0.841	ab	1766	223	51	0.920	0.959	0.888	ab	2045	283	0.894	0.935	0.878	
ade	286	59	31	0.774	0.827	0.829	acde	1431	183	50	0.745	0.777	0.887	acde	1749	242	0.764	0.799	0.878	
abe	286	60	31	0.773	0.827	0.827	ade	1424	180	49	0.742	0.773	0.888	ade	1710	239	0.747	0.782	0.877	
abde	286	60	31	0.773	0.827	0.827	acd	1346	182	50	0.699	0.731	0.881	acd	1657	240	0.723	0.757	0.873	
ae	284	59	31	0.768	0.821	0.828	ad	1339	179	49	0.696	0.727	0.882	ace	1603	157	0.710	0.733	0.911	
ad	279	58	30	0.754	0.806	0.828	ace	1287	98	50	0.682	0.699	0.929	ad	1618	237	0.705	0.739	0.872	
ab	279	60	30	0.753	0.806	0.823	ae	1276	95	49	0.676	0.693	0.931	ae	1560	154	0.691	0.713	0.910	
abd	279	60	30	0.753	0.806	0.823	bcde	1231	199	48	0.634	0.668	0.861	ac	1489	153	0.658	0.681	0.907	
a	274	58	30	0.740	0.792	0.825	bde	1225	198	48	0.631	0.665	0.861	a	1438	148	0.636	0.657	0.907	
bcde	94	6	20	0.266	0.272	0.940	ac	1183	95	49	0.626	0.642	0.926	bcde	1325	205	0.576	0.606	0.866	
bce	93	6	20	0.263	0.269	0.939	bce	1214	195	47	0.626	0.659	0.862	bce	1307	201	0.568	0.597	0.867	
bcd	87	6	19	0.246	0.251	0.935	be	1207	194	47	0.622	0.655	0.862	bde	1277	200	0.555	0.584	0.865	
bc	85	6	18	0.240	0.246	0.934	a	1164	90	46	0.616	0.632	0.928	be	1258	196	0.547	0.575	0.865	
cde	65	5	18	0.183	0.188	0.929	bcd	1056	199	47	0.539	0.573	0.841	bcd	1143	205	0.493	0.522	0.848	
ce	61	5	17	0.172	0.176	0.924	bd	1050	198	47	0.536	0.570	0.841	bc	1106	201	0.477	0.505	0.846	
cd	58	4	17	0.164	0.168	0.935	bc	1021	195	46	0.521	0.554	0.840	bd	1095	200	0.472	0.500	0.846	
bde	52	2	10	0.148	0.150	0.963	b	1013	194	46	0.517	0.550	0.839	b	1056	196	0.454	0.483	0.843	
be	51	2	10	0.146	0.147	0.962	cde	664	155	35	0.334	0.361	0.811	cde	729	160	0.310	0.333	0.820	
bd	45	2	9	0.128	0.130	0.957	de	645	151	33	0.324	0.350	0.810	de	668	152	0.283	0.305	0.815	
c	44	4	12	0.124	0.127	0.917	ce	477	59	30	0.249	0.259	0.890	ce	538	64	0.237	0.246	0.894	
b	43	2	8	0.122	0.124	0.956	cd	472	153	32	0.230	0.256	0.755	cd	530	157	0.220	0.242	0.771	
de	23	1	8	0.066	0.067	0.958	e	436	50	25	0.228	0.237	0.897	e	454	51	0.200	0.207	0.899	
e	18	1	6	0.051	0.052	0.947	d	453	148	28	0.221	0.246	0.754	d	469	148	0.193	0.214	0.760	
d	16	0	7	0.046	0.046	1.000	c	219	55	18	0.110	0.119	-	c	263	59	0.112	0.120	0.817	

MM - Main Metric. TM - Tiebreaker Metric. Rec. - Recall. Prec. - Precision. Pg - # of plugins

Infor. - Informedness. F-Meas. - F-Measure. Marked. - Markedness.

Tools: a - phpSAFE. b - RIPS. c - WAP. d - Pixy. e - WeVerca.

Table D.3.: Best Solutions for the WordPress plugins: SQLi, XSS and SQLi + XSS: Medium-quality

SQLi							XSS							SQLi + XSS						
Tools	TP	FP	Pg	MM	TM	P/R	Tools	TP	FP	Pg	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R	
Medium-quality							F-Meas.							F-Meas.						
abce	251	163	21	0.737	0.940	0.606	abce	2386	652	46	0.879	0.999	0.785	abcde	2637	815	0.863	0.993	0.764	
abcde	251	163	21	0.737	0.940	0.606	abcde	2386	652	46	0.879	0.999	0.785	abce	2637	815	0.863	0.993	0.764	
abc	250	163	21	0.735	0.936	0.605	abc	2383	652	46	0.879	0.998	0.785	abc	2633	815	0.863	0.991	0.764	
abcd	250	163	21	0.735	0.936	0.605	abcd	2383	652	46	0.879	0.998	0.785	abcd	2633	815	0.863	0.991	0.764	
abde	237	163	19	0.711	0.888	0.593	abde	2359	652	46	0.874	0.987	0.783	abde	2596	815	0.856	0.977	0.761	
abd	236	163	19	0.709	0.884	0.591	abe	2345	652	46	0.871	0.982	0.782	abe	2580	815	0.853	0.971	0.760	
abe	235	163	19	0.707	0.880	0.590	abd	2328	652	46	0.867	0.975	0.781	abd	2564	815	0.850	0.965	0.759	
ab	233	163	19	0.703	0.873	0.588	ab	2314	652	46	0.864	0.969	0.780	ab	2547	815	0.846	0.959	0.758	
acd	168	63	20	0.675	0.629	0.727	bcde	2006	494	44	0.821	0.840	0.802	bcde	2183	607	0.802	0.822	0.782	
ac	159	50	20	0.668	0.596	0.761	bce	1989	492	44	0.817	0.833	0.802	bce	2166	605	0.798	0.816	0.782	
acde	169	85	20	0.649	0.633	0.665	bde	1971	494	44	0.812	0.825	0.800	bde	2128	607	0.789	0.801	0.778	
bce	177	113	12	0.636	0.663	0.610	be	1938	491	44	0.805	0.811	0.798	be	2093	604	0.782	0.788	0.776	
bcde	177	113	12	0.636	0.663	0.610	bcd	1914	494	44	0.798	0.801	0.795	bcd	2090	607	0.781	0.787	0.775	
bc	176	113	12	0.633	0.659	0.609	bc	1891	491	43	0.793	0.792	0.794	bc	2067	604	0.776	0.778	0.774	
bcd	176	113	12	0.633	0.659	0.609	bd	1851	494	44	0.782	0.775	0.789	bd	2007	607	0.762	0.756	0.768	
ace	160	84	20	0.626	0.599	0.656	b	1812	490	43	0.773	0.759	0.787	b	1965	603	0.752	0.740	0.765	
ad	145	63	18	0.611	0.543	0.697	acde	1630	317	43	0.752	0.682	0.837	acde	1799	402	0.741	0.677	0.817	
ade	146	85	18	0.586	0.547	0.632	ade	1574	317	43	0.736	0.659	0.832	ade	1720	402	0.720	0.648	0.811	
bde	157	113	6	0.585	0.588	0.581	acd	1533	309	43	0.725	0.642	0.832	acd	1701	372	0.719	0.640	0.821	
bd	156	113	6	0.582	0.584	0.580	ace	1431	276	43	0.699	0.599	0.838	ace	1591	360	0.691	0.599	0.815	
be	155	113	6	0.579	0.581	0.578	ad	1435	309	43	0.694	0.601	0.823	ad	1580	372	0.686	0.595	0.809	
b	153	113	6	0.574	0.573	0.575	ae	1338	276	43	0.669	0.560	0.829	ae	1452	360	0.650	0.547	0.801	
ae	114	84	17	0.490	0.427	0.576	ac	1147	267	43	0.603	0.480	0.811	ac	1306	317	0.610	0.492	0.805	
cd	89	13	11	0.482	0.333	0.873	cde	1030	67	26	0.591	0.431	0.939	cde	1120	102	0.578	0.422	0.917	
a	99	50	15	0.476	0.371	0.664	de	962	65	25	0.563	0.403	0.937	de	1017	100	0.539	0.383	0.910	
cde	90	35	11	0.459	0.337	0.720	a	970	267	41	0.535	0.406	0.784	a	1069	317	0.529	0.402	0.771	
c	72	0	11	0.425	0.270	1.000	cd	828	59	24	0.506	0.347	0.933	cd	917	72	0.503	0.345	0.927	
ce	79	34	11	0.416	0.296	0.699	ce	786	24	23	0.491	0.329	0.970	ce	865	58	0.483	0.326	0.937	
d	54	13	4	0.323	0.202	0.806	d	717	56	23	0.454	0.300	0.928	d	771	69	0.441	0.290	0.918	
de	55	35	4	0.308	0.206	0.611	e	621	21	19	0.410	0.260	0.967	e	642	55	0.383	0.242	0.921	
e	21	34	3	0.130	0.079	0.382	c	344	13	18	0.251	0.144	0.964	c	416	13	0.270	0.157	0.970	

MM - Main Metric. TM - Tiebreaker Metric. Rec. - Recall. Prec. - Precision. Pg - # of plugins

Infor. - Informedness. F-Meas. - F-Measure. Marked. - Markedness.

Tools: a - phpSAFE. b - RIPS. c - WAP. d - Pixy. e - WeVerca.

Table D.4.: Best Solutions for the WordPress plugins: SQLi, XSS and SQLi + XSS: Low-quality

SQLi							XSS							SQLi + XSS						
Tools	TP	FP	Pg	MM	TM	P/R	Tools	TP	FP	Pg	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R	
Low-quality			Marked.	Prec.	Rec.		Marked.			Prec.	Rec.		Marked.			Prec.	Rec.			
bc	6	0	2	0.963	1.000	0.120	c	62	3	6	0.835	0.954	0.114	c	67	3	0.857	0.957	0.202	
bce	6	0	2	0.963	1.000	0.120	abce	543	117	12	0.822	0.823	0.996	cde	124	10	0.835	0.925	0.340	
bcde	6	0	2	0.963	1.000	0.120	abcde	543	117	12	0.822	0.823	0.996	ce	111	9	0.832	0.925	0.310	
bcd	6	0	2	0.963	1.000	0.120	abc	542	117	12	0.822	0.823	0.994	cd	87	8	0.819	0.916	0.252	
c	5	0	2	0.962	1.000	0.100	abcd	542	117	12	0.822	0.823	0.994	de	92	9	0.815	0.911	0.264	
ce	5	0	2	0.962	1.000	0.100	abde	534	116	12	0.818	0.822	0.980	e	73	8	0.802	0.901	0.216	
cde	5	0	2	0.962	1.000	0.100	abe	533	116	12	0.818	0.821	0.978	abcde	584	149	0.794	0.797	0.880	
cd	5	0	2	0.962	1.000	0.100	abd	533	116	12	0.818	0.821	0.978	abce	584	149	0.794	0.797	0.880	
be	1	0	1	0.959	1.000	0.020	ab	532	116	12	0.817	0.821	0.976	abc	583	149	0.794	0.796	0.879	
b	1	0	1	0.959	1.000	0.020	cde	119	10	9	0.816	0.923	0.218	abcd	583	149	0.794	0.796	0.879	
bd	1	0	1	0.959	1.000	0.020	acde	299	41	12	0.815	0.879	0.549	abde	571	148	0.789	0.794	0.869	
bde	1	0	1	0.959	1.000	0.020	ac	267	34	11	0.815	0.887	0.490	abe	570	148	0.788	0.794	0.868	
abce	41	32	8	0.554	0.562	0.820	ace	292	40	12	0.813	0.880	0.536	abd	570	148	0.788	0.794	0.868	
abc	41	32	8	0.554	0.562	0.820	ce	106	9	8	0.813	0.922	0.194	ab	569	148	0.788	0.794	0.867	
abcd	41	32	8	0.554	0.562	0.820	ade	283	40	12	0.808	0.876	0.519	bcde	427	92	0.788	0.823	0.767	
abcde	41	32	8	0.554	0.562	0.820	acd	276	39	12	0.806	0.876	0.506	bce	423	92	0.786	0.821	0.762	
ace	40	32	8	0.547	0.556	0.800	ae	273	39	12	0.804	0.875	0.501	bcd	412	92	0.780	0.817	0.750	
ac	40	32	8	0.547	0.556	0.800	a	244	33	10	0.803	0.881	0.448	d	51	7	0.776	0.879	0.156	
acd	40	32	8	0.547	0.556	0.800	de	92	9	9	0.799	0.911	0.169	bde	402	91	0.775	0.815	0.739	
acde	40	32	8	0.547	0.556	0.800	ad	260	38	12	0.798	0.873	0.477	bc	404	92	0.775	0.815	0.741	
abe	37	32	8	0.525	0.536	0.740	cd	82	8	9	0.797	0.911	0.150	be	397	91	0.773	0.814	0.733	
ab	37	32	8	0.525	0.536	0.740	bcde	421	92	11	0.787	0.821	0.772	acde	339	73	0.771	0.823	0.673	
abd	37	32	8	0.525	0.536	0.740	e	73	8	7	0.785	0.901	0.134	ace	332	72	0.768	0.822	0.665	
abde	37	32	8	0.525	0.536	0.740	bce	417	92	11	0.784	0.819	0.765	bd	387	91	0.767	0.810	0.721	
ae	36	32	7	0.517	0.529	0.720	bcd	406	92	11	0.777	0.815	0.745	ac	307	66	0.765	0.823	0.634	
a	36	32	7	0.517	0.529	0.720	bde	401	91	11	0.776	0.815	0.736	b	378	91	0.761	0.806	0.711	
ad	36	32	7	0.517	0.529	0.720	be	396	91	11	0.772	0.813	0.727	acd	316	71	0.760	0.817	0.644	
ade	36	32	7	0.517	0.529	0.720	bc	398	92	10	0.772	0.812	0.730	ade	319	72	0.760	0.816	0.647	
e	0	0	0	-2.000	-	0.000	bd	386	91	11	0.766	0.809	0.708	ae	309	71	0.755	0.813	0.634	
d	0	0	0	-2.000	-	0.000	b	377	91	10	0.760	0.806	0.692	ad	296	70	0.748	0.809	0.616	
de	0	0	0	-2.000	-	0.000	d	51	7	9	0.758	0.879	0.094	a	280	65	0.748	0.812	0.596	

MM - Main Metric. TM - Tiebreaker Metric. Rec. - Recall. Prec. - Precision. Pg - # of plugins

Infor. - Informedness. F-Meas. - F-Measure. Marked. - Markedness.

Tools: a - phpSAFE. b - RIPS. c - WAP. d - Pixy. e - WeVerca.

Table D.5.: Best Solutions for the WordPress plugins regardless scenarios: SQLi, XSS and XSS + XSS

SQLi						XSS						SQLi + XSS				
Tools	TP	FP	Pg	MM	TM	Tools	TP	FP	Pg	MM	TM	Tools	TP	FP	MM	TM
Rec. Prec.						Rec. Prec.						Rec. Prec.				
abce	675	260	74	0.915	0.722	abce	4935	1038	120	0.998	0.826	abde	5610	1298	0.987	0.812
abcde	675	260	74	0.915	0.722	abcde	4935	1038	120	0.998	0.826	abcde	5610	1298	0.987	0.812
abc	667	260	73	0.904	0.720	abde	4896	1036	120	0.990	0.825	ade	5527	1298	0.973	0.810
abcd	667	260	73	0.904	0.720	abe	4881	1034	120	0.987	0.825	acde	5527	1298	0.973	0.810
acde	592	181	73	0.802	0.766	abc	4860	1038	120	0.983	0.824	bde	5485	1296	0.965	0.809
abde	589	260	63	0.798	0.694	abcd	4860	1038	120	0.983	0.824	bcde	5468	1294	0.962	0.809
abe	587	260	63	0.795	0.693	abd	4793	1036	120	0.970	0.822	de	5374	1296	0.946	0.806
acd	584	158	72	0.791	0.787	ab	4777	1034	120	0.966	0.822	cde	5355	1294	0.942	0.805
ace	581	180	73	0.787	0.764	bcde	3789	822	114	0.766	0.822	abcd	4115	945	0.724	0.813
abd	581	260	62	0.787	0.691	bce	3750	813	113	0.759	0.822	abd	4092	748	0.720	0.845
ab	578	260	62	0.783	0.690	bde	3725	818	114	0.753	0.820	abe	4075	936	0.717	0.813
ac	570	145	72	0.772	0.797	be	3667	806	113	0.742	0.820	abce	3935	933	0.693	0.808
ade	497	181	61	0.673	0.733	acde	3500	567	116	0.708	0.861	bcd	3918	744	0.690	0.840
ad	489	158	60	0.663	0.756	bcd	3500	822	113	0.708	0.810	ae	3878	712	0.683	0.845
ae	463	180	60	0.627	0.720	bc	3427	811	109	0.693	0.809	ace	3874	921	0.682	0.808
a	438	145	57	0.594	0.751	ade	3421	563	115	0.692	0.859	bd	3818	945	0.672	0.802
bcde	326	123	41	0.442	0.726	bd	3408	818	113	0.689	0.806	be	3743	934	0.659	0.800
bce	325	123	41	0.440	0.725	b	3315	804	109	0.671	0.805	bce	3710	616	0.653	0.858
bcd	318	123	40	0.431	0.721	acd	3294	554	115	0.666	0.856	ce	3662	708	0.644	0.838
bc	316	123	39	0.428	0.720	ad	3173	550	114	0.642	0.852	e	3610	933	0.635	0.795
bde	210	115	17	0.285	0.646	ace	3129	436	116	0.633	0.878	acd	3512	919	0.618	0.793
cde	209	44	38	0.283	0.826	ae	3001	430	115	0.607	0.875	ad	3464	610	0.610	0.850
be	207	115	17	0.281	0.643	ac	2704	416	112	0.547	0.867	cd	3274	561	0.576	0.854
bd	202	115	16	0.274	0.637	a	2480	408	105	0.502	0.859	d	2918	553	0.514	0.841
cd	201	21	37	0.272	0.905	cde	1897	250	78	0.384	0.884	abc	2106	294	0.371	0.878
b	197	115	15	0.267	0.631	de	1777	241	75	0.359	0.881	bc	1855	277	0.326	0.870
ce	194	43	37	0.263	0.819	cd	1457	236	72	0.295	0.861	ab	1658	257	0.292	0.866
c	170	8	32	0.230	0.955	ce	1426	101	69	0.288	0.934	b	1620	144	0.285	0.918
de	78	36	12	0.106	0.684	d	1290	225	67	0.261	0.852	ac	1360	238	0.239	0.851
d	70	13	11	0.095	0.843	e	1174	84	58	0.238	0.933	a	1213	119	0.213	0.911
e	39	35	9	0.053	0.527	c	648	77	45	0.131	0.894	c	818	85	0.144	0.906

MM - Main Metric. TM - Tiebreaker Metric. Rec. - Recall. Prec. - Precision. Pg - # of plugins

Infor. - Informedness. F-Meas. - F-Measure. Marked. - Markedness.

Tools: a - phpSAFE. b - RIPS. c - WAP. d - Pixy. e - WeVerca.

Case Study: Synthetic Dataset Using the 1-out-of-n Strategy

The organization of the chapter is as follows. Section E.1 details the composition of the workload. Section E.2 presents the details about the benchmark run. Section E.3 presents an discuss the results. Section E.4 concludes the chapter. Section E.5 provides detailed results of the case study.

The goal of this experiment is to study the potential of combining the outputs of multiple SAST tools with a *1-out-of-n* strategy as a way to improve the performance of the vulnerability detection across different realistic development scenarios using a synthetic workload composed by small test cases. In practice, we formulate the following seven hypotheses, where the first four hypotheses are the same ones formulated in Section 5.1.1:

- H_1 : The number of vulnerabilities detected always increases as the number of combined SAST tools increases.
- H_2 : The number of FPs always increases as the number of combined SAST tools increases.
- H_3 : The best combination of SAST tools is the same across development scenarios.
- H_4 : The best combination of SAST tools is the same across different classes of vulnerabilities.
- H_5 : The best combination of SAST tools is the same regardless the classes of vulnerabilities.
- H_6 : The benchmark approach adopted can be used for other kind of applications (e.g., test cases with very small sizes).
- H_7 : The results for H_1 to H_5 are the same for any kind of application.

The contributions of this section are:

- 1) An experimental campaign with five free SAST tools to detect vulnerabilities in 19,632 PHP synthetic test cases from NIST;
- 2) A comparative experimental study combining the outputs of multiple (five) SAST tools using datasets with real applications (134 PHP Plugins) versus synthetic test cases.

E.1. Workload

The workload is the first component of the benchmark to be developed. Next sections detail the process used to compose the workload which, in our case, will be composed of synthetic test cases.

E.1.1. Collecting the Source Code of Vulnerable Applications

Synthetic test cases are snippets of code, complete programs, plugins, etc., that can be written by a programmer, extracted from production code or automatically generated by other programs. The use of programs to generate test cases has several advantages: the source code, vulnerable or not, is generated automatically according to a set of configuration settings; we know the test cases that contain vulnerabilities and the test cases that are not vulnerable; the characterization of the vulnerabilities is also generated automatically, including the class of vulnerability and the **location** in the source code.

The **SARD** at **NIST** already provides a repository of test cases (datasets) with a set of known security vulnerabilities [114]. The most significant synthetic dataset for PHP was provided by Stivalet and Delaitre [235]. They proposed a generic approach for generating *safe test cases* (i.e., non-vulnerable or safe sample) and *unsafe test case* (i.e., vulnerable or unsafe sample) and developed a tool (PHP Vulnerability Test Suite Generator¹), in **Python**, to generate test cases for PHP. Their test cases include six classes of vulnerabilities and are randomly distributed across 43 vulnerability directories. Since we are interested only in **SQLi** and **XSS** classes of vulnerabilities, we used the Stivalet’s tool to just generate test cases for them. To accomplish this, we ran the tool with the following command line argument, `-flaw=XSS, Injection`.

During the execution, we found some bugs in the application: That there are discrepancies in the number of the vulnerable **LOC** between the manifest file and the real number of vulnerable **LOC**. On 21/08/2017, we reported examples of the errors found in the manifest file to the responsible at **NIST** (samate@nist.gov, paul.black@nist.gov and charles.deoliveira@nist.gov). The response from **NIST**, on 29/08/2017, confirms the errors in the test suite “PHP Vulnerability Test Suite”² provided by Bertrand Stivalet and Aurelien Delaitre. In fact, the error is in the PHP Vulnerability Test Suite Generator of the test cases. Since the bug was not fixed by the developers in due time, we corrected automatically (using a small script in PHP to get the **LOC** where is located the **SS** function call) the number of the **LOC** in the test cases generated and used it in our study.

The test cases generated by the tool are constructed based on the specification of one *input* (**EP**), one *filtering* (**SF**) and one Sensitive Sink (**SS**) with one vulnerable variable. The test cases may include zero, one or several combined complexities expressed by adding control flow and/or data flow elements: `if`, `for`, `while`, `function`, `class` and `file`. Unfortunately, each test case targets only one flaw. As a result, the cases have a much simpler structure than most

¹<https://github.com/stivalet/PHP-Vuln-test-suite-generator>

²<https://samate.nist.gov/SRD/view.php?tsID=103>

vulnerabilities found in production software. Given this observation, we may consider these test cases as not representative of real software, nonetheless, they can be used to highlight weaknesses and strengths of **SAST** tools. Table E.1 shows the summary of the test cases generated and Table E.2 the test cases by scenario and class of vulnerability. The number of safe and unsafe test cases is not equal for both classes of vulnerabilities and it is very unbalanced for **SQLi**. In fact, this also occurs for real software (see Table 4.8). A common way to create an unsafe test case is to inject a vulnerability in a safe test case. In this way, we can create workloads with an equal number of safe and unsafe test cases related to each other. This is not the case for the generated test cases. Therefore, the safe test cases and the unsafe test cases are not related, meaning that for all unsafe test case there is no the corresponding safe test case (i.e., a version of the unsafe test case fixed). For instance, it is not possible the use of the *DR* metric proposed by **OWASP** for evaluating security tools (see Section 2). For classifying the test cases as **OOP** or **POP**, we used the tool **PHPdepend**. The tool outputs a **XML** file with several **SCMs** about the source code, including the number of classes and number of **LOC** in classes. If a test case has definition of one or more classes, then the test case is classified as **OOP** otherwise is classified as **POP**.

Table E.1.: Summary of the synthetic test cases the generated.

Class	Safe				Unsafe				Total
	POP	OOP	Total	%Total	POP	OOP	Total	%Total	
SQLi	6,336	2,304	8,640	90.0	684	228	912	10.0	9,552
XSS	4,296	1,432	5,728	57.0	3,264	1,088	4,352	43.0	10,080
Total	10,632	3,736	14,368	73.2	3,948	1,316	5,264	26.8	19,632

Table E.2.: Synthetic test cases background information by scenario and class of vulnerability.

Scenario	SQLi						XSS					
	POP		OOP		Total		POP		OOP		Total	
	N	P	N	P	N	P	N	P	N	P	N	P
1 - Highest-quality	1,908	213	228	2,208	2,136	2,421	3,880	2,940	1,432	1,088	5,312	4,028
2 - High-quality	1,143	195	0	96	1,143	291	324	236	0	0	324	236
3 - Medium-quality	3,285	276	0	0	3,285	276	92	88	0	0	92	88
4 - Low-quality	0	0	0	0	0	0	0	0	0	0	0	0
Total	6,336	684	228	2,304	6,564	2,988	4,296	3,264	1,432	1088	5,728	4,352

Listing E.1 shows an example of a generated safe test case for **SQLi**. It includes an example of user input data. The data coming from the `$_GET['userData']` **EP** is passed to an array and then to the variable `$tainted`. Next, the variable is sanitized via `filter_var`³ function in lines 6 and 7. The option `FILTER_SANITIZE_MAGIC_QUOTES` sets backslashes in front of predefined characters (`'`, `"`, `\` and `NULL`) to avoid characters that could lead to an unintended **SQL** command. For a user input: `"Users'; DROP TABLE Users; --"`, the value of the `$query` variable becomes: `"SELECT * FROM '\ ' Users'; DROP TABLE Users; --"`. Once the variable has been sanitized, the line 12 it is no longer vulnerable to **SQLi**. For this test case, all **SAST** tools did not report any vulnerability at line 12, which is correct, since it does not exist.

Listing E.2 shows an example of a generated unsafe test case for **SQLi** and Figure E.1 shows the corresponding data flow from the **EP** do to **SS**. Data coming from the `$_GET['userData']`

³<https://www.php.net/manual/en/filter.filters.sanitize.php>

```

1  $array = array();
2  $array[] = 'safe' ;
3  $array[] = $_GET['userData'] ;
4  $array[] = 'safe' ;
5  $tainted = $array[1] ;
6  $sanitized = filter_var($tainted, FILTER_SANITIZE_MAGIC_QUOTES);
7  $tainted = $sanitized ;
8  $query = "SELECT * FROM '$tainted'";
9  $conn = mysql_connect('localhost', 'mysql_user', 'mysql_password'); // Connection to the database (address,
    user, password)
10 mysql_select_db('dbname') ;
11 echo "query : ". $query ."<br /><br />" ;
12 $res = mysql_query($query); //execution
13 while($data =mysql_fetch_array($res)){
14     print_r($data) ;
15     echo "<br />" ;
16 }
17 mysql_close($conn);

```

Listing E.1: Example of safe test case generated [SQLi](#).

```

1  $array = array();
2  $array[] = 'safe' ;
3  $array[] = $_GET['userData'];
4  $array[] = 'safe' ;
5  $tainted = $array[1] ;
6  //no_sanitizing
7  $query = "SELECT * FROM '$tainted'";
8  //flaw
9  $conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
10 mysql_select_db('dbname') ;
11 echo "query : ". $query ."<br /><br />" ;
12 $res = mysql_query($query); //execution
13 while($data = mysql_fetch_array($res)){
14     print_r($data) ;
15     echo "<br />" ;
16 }
17 mysql_close($conn);

```

Listing E.2: Example of a generated unsafe test case for [SQLi](#).

[EP](#) is passed to an array and then to the variable `$tainted`. The variable is used without any sanitization to build the [SQL](#) query to be executed in the `mysql_query` [SS](#). Therefore, the test case is vulnerable to [SQLi](#) in line 12. For this test case, the [SAST](#) tools `phpSAFE`, `RIPS` and `WAP` did not report the [SQLi](#) vulnerability, but `Pixy` and `WeVerca` did it correctly.

A key observation for the unsafe test cases generated for a target class of vulnerabilities (e.g., [SQLi](#)) can also have other classes of vulnerabilities (e.g., [XSS](#)). For example, the target class of the test case in [Listing E.2](#) is [SQLi](#), but it also includes a [XSS](#) vulnerability. In fact, the line 11 is vulnerable to [XSS](#), but this was not included in the manifest file. Therefore, the [SAST](#) tools have to be configured to detect vulnerabilities in the target class of the test case and, when the tool does not allow this configuration, the results for other classes of vulnerabilities must be ignored.

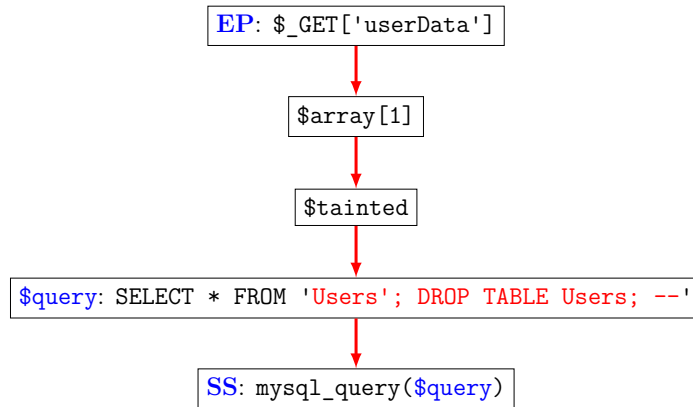


Figure E.1.: Data flow for the unsafe test case in Listing E.2, with an example of user input data. EP: PHP GET array, SS mysql_query.

E.1.2. Assigning Test Cases to Scenarios

For assigning the test cases to scenarios we used the methodology “Assigning Applications to Scenarios” described in Section 4.1.3.2. It requires the measures of the **SCMs** of the source code of the test cases. To gather the measures we used the same tools pointed out in Subsection 4.2.2 (PHPdepend, SonarQube and PHPMD).

The results of applying the methodology are presented in Table E.2. By taking a look at the table, we can see that most of the test cases were assigned to the highest-quality scenario where almost of the test cases are **OOP**. The high-quality scenario accounts for 10%, the medium-quality scenario 19%, and there are none in the low-quality scenario. This distribution occurred because the test cases are very small programs (from 3 **LLOC** to 29 **LLOC**). Therefore, the values for some **SPPs** (e.g., Module Coupling) of the test cases are zero, meaning that for these **SPPs** the test cases have the maximum quality (5 stars).

E.1.3. Characterizing VLOCs and NVLOCs of Synthetic Test Cases

An advantage of Synthetic Datasets (**SDs**) is that for each test case it is indicated the location where the vulnerabilities occur. All test cases generated by the PHP Vulnerability Test Suite Generator tool have just one target **SS** for the class of vulnerability. For instance, the number of **NVLOC** is equal to the number of safe test cases (column N of Table E.2) and the number of **VLOCs** is equal to the number of unsafe test cases (column P of Table E.2). For every class of vulnerability, the generator produces a manifest file, called `manifest.xml`. Its role is to keep track of every test case that has been produced and make the analysis easier. For each test case, it is generated an entry in the manifest file with the attributes described in Table E.3. If the test case is unsafe, a tag `<flaw>` is added to indicate the vulnerability line and the class of vulnerability.

Listing E.3 illustrates the entry in the manifest file for the unsafe test case present in Listing E.2. It includes the **EP** in the `<input>` tag, the path and the filename of the code in the attribute `path` of the `<file>` tag. The filename follows a specific pattern: `CWE_[CWE_number]_[Input]_[Filtering]`

Table E.3.: Manifest XML tags description.

Tags	Description
<testcase>	Description of a sample
<metadata>	Contains information about the sample
<file>	Describes the generated test case with its path and its language

```

1 <testcase>
2   <meta-data>
3     <author>Bertrand STIVALET, Aurelien DELAITRE</author>
4     <date>04/08/17</date>
5     <input>variable : $_GET['userData']</input>
6   </meta-data>
7   <file path="CWE_89/unsafe/CWE_89__array-GET__no_sanitizing__select_from-interpretation_simple_quote.php"
      language="PHP">
8     <flaw line="12" name ="Injection"/>
9   </file>
10 </testcase>

```

Listing E.3: Manifest file for the unsafe test case in Listing E.1.

_[Sink].php. The parts of filename of the example means:

- `CWE_89`: [CWE](#) vulnerability class (i.e., [SQLi](#)).
- `array-GET`: input (i.e., [EP](#)), PHP GET array (line 3).
- `no_sanitizing`: no filtering.
- `select_from-interpretation_simple_quote.php`: [SS](#): “SELECT FROM” with construction: concatenation with simple quote (line 7).

The location and class of vulnerability is detailed in the `flaw` tag. The manifest file does not include the name of the [SS](#). For the test case in Listing E.2, the name is `mysql_query`. Therefore, for gathering the number of the [LOC](#) and name of the [SSs](#) for the safe and unsafe test cases, we used the same process explained in Section 4.2.3. Thus, the PHP script described in that section, was executed for all test cases for gathering all [SSs](#) function calls.

E.2. Benchmark Run

To run the benchmark we follow the benchmarking procedure defined in Section 4.1.4, which is detailed next:

- 1) **Preparation:** In this step of the benchmark, we have to select the [SAST](#) tools to be benchmarked and to specify the configuration settings for the tools. We used the same [SAST](#) tools with the same configurations settings used in the benchmarking experiment for WordPress plugins (see Chapter 4.2) because we want to compare the results between the two benchmarking experiments. Like for in the benchmarking experiment for WordPress plugins, we automated the process of running the [SAST](#) tools by writing shell scripts, based on the list of all source code files in the workload, to run the tools, analyzing one file and one class of vulnerability per run and storing the results in a separate file.

- 2) **Execution:** We ran the benchmark for all the [SAST](#) tools, searching for [XSS](#) and [SQLi](#) vulnerabilities in the workload composed by the synthetic test cases. The tools successfully analyzed all test cases.
- 3) **Normalization of Reports:** The vulnerability reports of the [SAST](#) tools were normalized to a common format according to the structure presented in [Table C.1-Common format for the results of SAST tools](#).
- 4) **Vulnerability Verification:** The reports of the [SAST](#) tools are in the same format of the list of [VLOCs](#) and [NVLOCs](#) of the workload. Therefore, the correctness of the vulnerabilities reported by the tools was automatically verified by matching the normalized results of the tools with the list of [VLOCs](#) and the list of [NVLOC](#).
- 5) **Metrics Calculation and Ranking:** The benchmark metrics are calculated automatically based on the [SAST](#) tools outputs and their verification, the number of P and N instances by scenario and class of vulnerability (see [Table E.2](#)). Afterwards, [SAST](#) tools are ranked according to the metrics recommend for each scenario (see [Table 4.1](#)).

E.3. Results and Discussion

[Table E.4](#) list the results organized by scenario, class of vulnerability and together [SQLi](#) and [XSS](#) vulnerabilities for the [SD](#). [Table E.4](#) only shows the TOP 5 (of 31) combinations of [SAST](#) tools for each scenario, and the complete results can be found in [Appendix E.5-Best Solutions for the Synthetic Dataset](#) and are also available online at [\[194\]](#). The results for the [WPD](#) are depicted in [Table 5.1](#) of [Section 5.1](#). [Tables 5.1](#) and [E.4](#) for all scenarios include the metrics *Recall* and *Precision* to facilitate the determination of the scenario-specific impact. The tables also include the ranking of the individual [SAST](#) tools, as reference.

E.3.1. Comparing the Results of the WordPress Plugins Dataset and the Synthetic Dataset

The best solutions for both datasets are very different. For the [WPD](#), most of the solutions are composed by several [SAST](#) tools, while for the [SD](#), most of the solutions consist of a single [SAST](#) tool, except for the highest-quality scenario.

As an overall remark, for the [SD](#) the [SAST](#) tools reported few [TPs](#) and many [FPs](#), including [WAP](#) which for the [WPD](#) reported very few [FPs](#). In fact, the effectiveness of the [SAST](#) tools is better for [WPD](#) than for [SD](#). As a consequence, for the [SD](#) and for the high-quality and medium-quality scenarios, the number of [SAST](#) for the best solutions is only one, while for the [WPD](#) the number of combined [SAST](#) is four.

The overall results show that the values of the main metrics of all best solutions for the [SD](#) are lower than for the [WPD](#), meaning that the individual effectiveness of the [SAST](#) tools is much lower for the [SD](#). For instance, in several cases the worst individual tool (e.g., [Pixy](#)) for the [WPD](#) is the best individual tool for the [SD](#). The inverse occurred for other tools (e.g., [WAP](#) for

Table E.4.: Best Solutions for the synthetic dataset: SQLi, XSS and SQLi + XSS

SQLi						XSS						SQLi + XSS		
Tools	TP	FP	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R	Tools	MM	TM
Highest-quality			Recall	Prec.		Recall			Prec.		Recall			Prec.
bde	355	2072	0.805	0.146	-	abde	3290	4192	0.817	0.440	-	abde	0.816	0.368
abde	355	2072	0.805	0.146	-	abcde	3290	4232	0.817	0.437	-	abcde	0.816	0.364
bcde	355	2132	0.805	0.143	-	ade	3286	4142	0.816	0.442	-	ade	0.812	0.378
abcde	355	2132	0.805	0.143	-	acde	3286	4182	0.816	0.440	-	acde	0.812	0.374
de	342	1607	0.776	0.176		bde	3226	4176	0.801	0.436	-	bde	0.801	0.364
e	234	1086	0.531	0.177	-	e	2336	3209	0.580	0.421	-	e	0.575	0.374
d	156	609	0.354	0.204	-	d	1958	1783	0.486	0.523	-	d	0.473	0.469
b	126	885	0.286	0.125	-	b	1048	1436	0.260	0.422	-	b	0.263	0.336
a	75	468	0.170	0.138	-	a	656	864	0.163	0.432	-	a	0.164	0.354
c	63	430	0.143	0.128	-	c	408	712	0.101	0.364	-	c	0.105	0.292
High-quality		Informedness		Rec.	Prec.	Informedness		Rec.	Prec.		Inf0.		Prec.	
d	150	222	0.590	0.769	0.403	d	121	107	0.183	0.513	0.531	d	0.418	0.629
de	190	498	0.572	0.974	0.276	cd	121	107	0.183	0.513	0.531	de	0.387	0.877
bd	183	504	0.532	0.939	0.266	ad	121	107	0.183	0.513	0.531	bd	0.367	0.861
cde	190	597	0.493	0.974	0.241	acd	121	107	0.183	0.513	0.531	bcd	0.332	0.861
bcd	183	558	0.488	0.939	0.247	c	0	0	0.000	0.000	-	cd	0.332	0.629
d	150	222	0.590	0.769	0.403	d	121	107	0.183	0.513	0.531	d	0.418	0.629
e	128	376	0.353	0.656	0.254	a	0	0	0.000	0.000	-	e	0.260	0.657
b	81	372	0.115	0.415	0.179	c	0	0	0.000	0.000	-	b	0.215	0.624
a	81	438	0.062	0.415	0.156	b	188	268	-0.031	0.797	0.412	c	-0.037	0.084
c	36	188	0.033	0.185	0.161	e	155	244	-0.096	0.657	0.388	a	-0.092	0.188
Medium-quality		F-Measure		Rec.	Prec.	F-Measure		Rec.	Prec.		F-Measure		Rec.	
e	132	882	0.205	0.478	0.130	d	81	90	0.626	0.921	0.474	e	0.243	0.484
ce	132	882	0.205	0.478	0.130	ad	81	90	0.626	0.921	0.474	ce	0.243	0.484
ae	132	948	0.195	0.478	0.122	bd	81	90	0.626	0.921	0.474	ace	0.232	0.484
ace	132	948	0.195	0.478	0.122	cd	81	90	0.626	0.921	0.474	ae	0.232	0.484
de	265	2588	0.169	0.96	0.093	de	81	90	0.626	0.921	0.474	cde	0.204	0.951
e	132	882	0.205	0.478	0.130	d	81	90	0.626	0.921	0.474	e	0.243	0.484
d	225	2310	0.160	0.815	0.089	e	44	28	0.550	0.500	0.611	d	0.199	0.841
b	105	1155	0.137	0.380	0.083	a	0	0	0.000	0.000	0.000	b	0.129	0.288
a	0	66	0.000	0.000	0.000	b	0	0	0.000	0.000	0.000	c	-	0.000
c	0	0	0.000	0.000	-	c	0	0	0.000	0.000	0.000	a	-	0.000
*bde	824	5621	-	0.128 ¹	0.904 ²	*abde	3559	4588	-	0.437 ¹	0.818 ²	*de	0.447 ¹	0.573 ²

MM - Main Metric, TM - Tiebreaker Metric, Rec. - Recall (R)⁽²⁾, Prec.(P)⁽¹⁾ - Precision

Tools: a - phpSAFE, b - RIPS, c - WAP, d - Pixy, e - WeVerca. Inf. - Informedness. *See table 4.10

the highest-quality scenario). Pixy is old but was included in five out of six best solutions using the SD. Moreover, the tool is the best solution for the SD in three cases. As stated before, the tool is limited when analyzing OOP. Therefore, the good results of Pixy for these three cases are due to the presence of POP in most of the test cases. It means that for the SD plugins, Pixy has a good effectiveness analyzing POP test cases and the other tools have low effectiveness because they fail analyzing these simple POP test cases. For example, phpSAFE, RIPS and WAP did not report any vulnerability in several cases (see Table E.4). For the WPD, Pixy was only included in one out of eight best solutions. This occurred because in this scenario about 1/3 of the plugins are POP and 49% of the LLOC are POP coming both from the POP plugins and OOP with POP code.

We investigated some reasons of the detection failure. From the 912 unsafe test cases for [SQLi](#), there are 57 with an [EP](#), as for example, the test case of Listing [E.2](#), lines 1 to 5. For instance, in this test case, the [SAST](#) tools [phpSAFE](#), [RIPS](#) and [WAP](#) do not report the variable `$tainted` as vulnerable. The PHP “[]” operator appends data to an array. The data is stored in an integer index calculated as the highest integer index of the array plus 1. In fact, the [SAST](#) tools have serious limitations modeling arrays. In contrast, [Pixy](#) reported vulnerabilities in 51 test cases and [WeVerca](#) in 38. Since the arrays are commonly used in PHP applications, these [SAST](#) tools miss many vulnerabilities. This is another important indication for improvement for the developers of these [SAST](#) tools.

For the WordPress Plugins Dataset ([WPD](#)), [WAP](#) reports the fewest [FPs](#) (high precision) and is included in seven out of eight solutions. However, for the Synthetic Dataset ([SD](#)), [WAP](#) has low precision and is never included in the best solutions. A possible reason is that [WAP](#) uses data mining to identify [FPs](#) using a machine learning classifier. Perhaps, the classifier has to be better trained for [SDs](#).

For the [WPD](#), [WeVerca](#) is ranked in the middle or below for all cases. However, for the [SD](#), the tool was ranked first or second for all cases, except for the [XSS](#) and for the high-quality scenario where it was ranked in the last position. The tool reported both the highest number of [TPs](#) and [FPs](#) for most of the scenarios and classes of vulnerabilities. Therefore, the tool needs to be improved in order to report less [FPs](#).

For the highest-quality scenario, the values of the main metrics are similar for both datasets. In contrast, the values for the tiebreaker metrics are very low for the [SD](#) (e.g., 0.146 for [SQLi](#)) and high for the [WPD](#) (e.g., 0.929 for [SQLi](#)). Therefore, the tools have better effectiveness for the [WPD](#) than for the [SD](#).

For both datasets and considering [SQLi](#) and [XSS](#) vulnerabilities together, the best solution includes the [SAST](#) tools of the best solution for each class of vulnerability, with two exceptions. First, for the [WPD](#) and for the low-quality scenario (see Table [5.1](#)), the best solution excludes [RIPS](#) because it reported many [FPs](#) for [XSS](#) and only one [TP](#) for [SQLi](#). Second, for the [SD](#) and for the medium-quality scenario (see Table [E.4](#)), the best solution is the same as for [XSS](#) which excludes the [SAST](#) tool of the best solution for [SQLi](#). This may mask the effectiveness of the solutions for a specific class of vulnerability. For example, comparing the overall results ([SQLi+XSS](#)) for the [WPD](#) with the results for [SQLi](#) we saw differences for the main metric ranging from -0.105 to 0.126. The maximum difference, for the main metric, occurs for the *medium-quality scenario*, with 0.737 for [SQLi](#), 0.879 [XSS](#) and 0.863 for [SQLi+XSS](#).

E.3.2. Testing the Hypotheses

Based on our findings, we conclude that the first five hypotheses stated in the introduction ([E](#)) are false and hypotheses the H_6 and H_7 are not false:

- Hypothesis H_1 (the number of vulnerabilities detected always increases as the number of combined [SAST](#) tools increases) is false, because we found many cases where adding a [SAST](#) tool to an existing combination of [SAST](#) tools, does not increase the number of

vulnerabilities found (e.g., for the highest-quality scenario and **XSS**: *ab, abe, abce*, see Table 5.1). On the other hand, we also observed that the number of **FPs** does not always increase with the number of **SAST** tools in a combination (e.g., for the plugins, the medium-quality scenario and **SQLi**: *abc, abcd, abcde*, See Table 5.1). As there is frequently an overlap between the **FPs** reported by different **SAST** tools, in some cases, having a combination with more tools can detect more vulnerabilities, while maintaining the same number of **FPs**. Also note that, none of the best combinations includes all **SAST** tools.

- On the other hand, we also observed that the number of **FPs** does not always increase with the number of **SAST** tools in a combination (e.g., for the medium-quality scenario and **SQLi**: *ab, abe, abde*). Therefore, hypothesis H_2 (the number of false positives always increases as the number of combined **SAST** tools increases) is also false. As there is frequently an overlap between the **FPs** reported by different **SAST** tools, in some cases combinations with more tools can detect more vulnerabilities, while maintaining the same number of **FPs**. Also note that, none of the best combinations includes all **SAST** tools.
- The best solution for vulnerability detection depends on the chosen scenario and on class of vulnerability. Therefore, hypotheses H_3 (the best combination of **SAST** tools is the same across development scenarios) and H_4 (the best combination of **SAST** tools is the same across different classes of vulnerabilities) are both false. In fact, the detection capabilities of the **SAST** tools are not uniform across the two classes of vulnerabilities. The same occurs for combinations of **SAST** tools. Moreover, in almost all cases the values of the metrics for **XSS** vulnerabilities are better than for the **SQLi** vulnerabilities. The best combination of **SAST** tools regardless the classes of vulnerabilities is different in several cases. Therefore, the hypothesis H_5 (the best combination of **SAST** tools is the same regardless the classes of vulnerabilities) is also false.
- The approach was successfully applied to another kind of dataset with similar number of **P** and **N** instances but with applications (the **NIST** test cases) with very small sizes (i.e., **LLOC**). Therefore, the hypothesis H_6 (The approach can be used for any kind of application) is not false.
- The results for the **SD** show that we can derive similar conclusions for the hypotheses H_1 to H_5 . Therefore, the hypothesis H_7 (The results for the hypotheses H_1 to H_5 are the same for other kind of applications) is not false.

In summary, the main advantage of combining the results of several **SAST** tools (the right collection of tools for the target scenario) is the identification of more vulnerabilities. In fact, for several cases there are **SAST** tools that individually did not find any vulnerabilities or found few vulnerabilities in many plugins. Moreover, even using all the **SAST** tools some vulnerabilities remain undetected. However, combining many tools can be counterproductive in some cases as that will not lead to the detection of more vulnerabilities, but will increase the number of **FPs** reported, which then need to be verified manually by the developers. Finally, identifying the strengths and limitations of **SAST** tools, helps developers to determinate how such tools can be combined to provide a more thorough analysis of the software depending on the specificities of the scenario and on the class of vulnerability being analyzed.

E.4. Conclusion

In this section, first we instantiate our benchmark approach for a synthetic workload composed by PHP synthetic test cases. Then, we addressed the problem of combining the output of several **SAST** tools searching for **SQLi** and **XSS** vulnerabilities in two different datasets, one with WordPress plugins and another with PHP synthetic test cases. The datasets were organized in four scenarios of increasing criticality and each scenario used different metrics to rank the tools.

Our findings revealed that combining the outputs of several free **SAST** tools do not always improve the vulnerability detection rate. Thus, the best solution can be a single tool or a combination of tools that may not include all the tools under evaluation. In principle, combining multiple **SAST** tools has benefits due to the complementarity of the produced results. However, for solutions including **SAST** tools that report many **FPs** the overall performance is worse in some scenarios.

The results of our comparative evaluation among the two case studies showed very significant performance differences of the **SAST** tools. The best **SAST** tool for one dataset was the worst **SAST** tool for the other dataset and vice-versa. Moreover, there was a considerable discrepancy on both the number of **TPs** and **FPs** among the combined **SAST** tools. However, the results for the **SD** revealed to be very usefully for developers in order for them to improve their tools and end-users choosing their **SAST** tools. Thus, due to the simplicity of the test cases, it is easy to identify the types of code constructs where the tools miss vulnerabilities and report **FPs**.

Our results highlighted a considerable variance on the rates of **TPs** and **FPs** among the **SAST** tools and the datasets. This means that, overall, the best combination of **SAST** tools is highly dependable on the specific situation, and it should be selected after a properly targeted benchmarking procedure, such as ours.

These results are very useful to help software engineers choosing a combination of **SAST** tools for a concrete project with a particular criticality and for **SAST** tools developers improving their tools. There are standard ways to evaluate the performance of **SAST** tools (e.g., [167] and [117]). However, they lack datasets in the domain of the software in production, fully characterized in terms of **VLOCs** and **NVLOCs**. Thus, we expect that our results encourage other researchers and software houses performing similar studies based on our methodology, producing datasets in diverse application domains. These datasets will be a support for developing new benchmarks to systematically evaluate the performance of **SAST** tools.

E.5. Best Solutions for the Synthetic Dataset

Tables D.1 to D.5 lists all results of combining the results of five **SAST** tools analyzing a synthetic dataset using *1-out-of-n* strategy (see Appendix E-Case Study: Synthetic Dataset Using the 1-out-of-n Strategy for more details). The tables contains data results as follows:

- Table E.5: Highest-quality.
- Table E.6: High-quality.

- Table E.7: Medium-quality.
- Table E.8: Regardless scenarios.

Table E.5.: Best Solutions for the synthetic dataset: XSS and SQLi + XSS: Highest-quality

SQLi						XSS						SQLi + XSS						
Tools	TP	FP	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R	
Highest-quality			Rec.	Prec.					Rec.	Prec.					Rec.	Prec.		
bde	355	2072	0.805	0.146	-	abde	3290	4192	0.817	0.440		abde	3645	6264	-	0.816	0.368	-
abde	355	2072	0.805	0.146	-	abcde	3290	4232	0.817	0.437		abcde	3645	6364	-	0.816	0.364	-
bcde	355	2132	0.805	0.143	-	ade	3286	4142	0.816	0.442		ade	3628	5971	-	0.812	0.378	-
abcde	355	2132	0.805	0.143	-	acde	3286	4182	0.816	0.440		acde	3628	6071	-	0.812	0.374	-
de	342	1607	0.776	0.176	-	bde	3226	4176	0.801	0.436		bde	3581	6248	-	0.801	0.364	-
cde	342	1817	0.776	0.158	-	bcde	3226	4216	0.801	0.434		bcde	3581	6348	-	0.801	0.361	-
ade	342	1829	0.776	0.158	-	de	3210	3950	0.797	0.448		de	3552	5557	-	0.795	0.390	-
acde	342	1889	0.776	0.153	-	cde	3210	4111	0.797	0.439		cde	3552	5928	-	0.795	0.375	-
be	262	1635	0.594	0.138	-	abcd	2525	2908	0.627	0.465		abcd	2744	4366	-	0.614	0.386	-
abe	262	1635	0.594	0.138	-	abd	2483	2670	0.616	0.482		abd	2702	4020	-	0.605	0.402	-
bce	262	1695	0.594	0.134	-	bcd	2461	2892	0.611	0.460		abe	2689	5093	-	0.602	0.346	-
abce	262	1695	0.594	0.134	-	abe	2427	3458	0.603	0.412		abce	2689	5195	-	0.602	0.341	-
ce	249	1372	0.565	0.154	-	abce	2427	3500	0.603	0.410		bcd	2680	4350	-	0.600	0.381	-
ae	249	1390	0.565	0.152	-	bd	2419	2654	0.601	0.477		ae	2666	4798	-	0.597	0.357	-
ace	249	1450	0.565	0.147	-	ae	2417	3408	0.600	0.415		ace	2666	4900	-	0.597	0.352	-
e	234	1086	0.531	0.177	-	ace	2417	3450	0.600	0.412		bd	2638	4004	-	0.590	0.397	-
bd	219	1350	0.497	0.140	-	be	2363	3442	0.587	0.407		be	2625	5077	-	0.587	0.341	-
abd	219	1350	0.497	0.140	-	bce	2363	3484	0.587	0.404		bce	2625	5179	-	0.587	0.336	-
bcd	219	1458	0.497	0.131	-	e	2336	3209	0.580	0.421		ce	2585	4750	-	0.578	0.352	-
abcd	219	1458	0.497	0.131	-	ce	2336	3378	0.580	0.409		e	2570	4295	-	0.575	0.374	-
ad	168	939	0.381	0.152	-	acd	2272	2549	0.564	0.471		acd	2440	3596	-	0.546	0.404	-
acd	168	1047	0.381	0.138	-	ad	2230	2311	0.554	0.491		ad	2398	3250	-	0.537	0.425	-
d	156	609	0.354	0.204	-	cd	2030	2231	0.504	0.476		cd	2186	3146	-	0.489	0.410	-
cd	156	915	0.354	0.146	-	d	1958	1783	0.486	0.523		d	2114	2392	-	0.473	0.469	-
b	126	885	0.286	0.125	-	abc	1214	1750	0.301	0.410		abc	1340	2743	-	0.300	0.328	-
ab	126	885	0.286	0.125	-	bc	1150	1734	0.286	0.399		bc	1276	2727	-	0.286	0.319	-
bc	126	993	0.286	0.113	-	ab	1112	1452	0.276	0.434		ab	1238	2337	-	0.277	0.346	-
abc	126	993	0.286	0.113	-	b	1048	1436	0.260	0.422		b	1174	2321	-	0.263	0.336	-
a	75	468	0.170	0.138	-	ac	758	1162	0.188	0.395		ac	833	1738	-	0.186	0.324	-
ac	75	576	0.170	0.115	-	a	656	864	0.163	0.432		a	731	1332	-	0.164	0.354	-
c	63	430	0.143	0.128	-	c	408	712	0.101	0.364		c	471	1142	-	0.105	0.292	-

MM - Main Metric. TM - Tiebreaker Metric. Rec. - Recall. Prec. - Precision.

Infor. - Informedness. F-Meas. - F-Measure. Marked. - Markedness.

Tools: a - phpSAFE. b - RIPS. c - WAP. d - Pixy. e - WeVerca.

Table E.6.: Best Solutions for the synthetic dataset: XSS and SQLi + XSS: High-quality

SQLi						XSS						SQLi + XSS					
Tools	TP	FP	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R
High-quality			Infor.	Rec.	Prec.				Infor.	Rec.	Prec.				Infor.	Rec.	Prec.
d	150	222	0.590	0.769	0.403	d	121	107	0.182	0.513	0.531	d	271	329	0.418	0.629	0.452
de	190	498	0.572	0.974	0.276	cd	121	107	0.182	0.513	0.531	de	378	766	0.387	0.877	0.330
bd	183	504	0.532	0.939	0.266	ad	121	107	0.182	0.513	0.531	bd	371	772	0.367	0.861	0.325
cde	190	597	0.493	0.974	0.241	acd	121	107	0.182	0.513	0.531	bcd	371	826	0.332	0.861	0.310
bcd	183	558	0.488	0.939	0.247	c	0	0	0.000	0.000	-	cd	271	464	0.332	0.629	0.369
cd	150	357	0.481	0.769	0.296	a	0	0	0.000	0.000	-	abd	371	838	0.325	0.861	0.307
ad	183	570	0.478	0.939	0.243	ac	0	0	0.000	0.000	-	cde	378	865	0.324	0.877	0.304
abd	183	570	0.478	0.939	0.243	b	188	268	-0.031	0.797	0.412	abcd	371	892	0.290	0.861	0.294
acd	183	624	0.435	0.939	0.227	ab	188	268	-0.031	0.797	0.412	ad	304	677	0.272	0.705	0.310
abcd	183	624	0.435	0.939	0.227	bc	188	268	-0.031	0.797	0.412	e	283	620	0.260	0.657	0.313
bde	195	712	0.425	1.000	0.215	bd	188	268	-0.031	0.797	0.412	acd	304	731	0.238	0.705	0.294
bcde	195	742	0.401	1.000	0.208	de	188	268	-0.031	0.797	0.412	bde	383	1018	0.237	0.889	0.273
ade	195	778	0.372	1.000	0.200	cde	188	268	-0.031	0.797	0.412	ade	383	1046	0.219	0.889	0.268
abde	195	778	0.372	1.000	0.200	ade	188	268	-0.031	0.797	0.412	bcde	383	1048	0.218	0.889	0.268
e	128	376	0.353	0.656	0.254	abd	188	268	-0.031	0.797	0.412	b	269	640	0.215	0.624	0.296
acde	195	808	0.348	1.000	0.194	bcd	188	268	-0.031	0.797	0.412	ce	295	742	0.210	0.684	0.284
abcde	195	808	0.348	1.000	0.194	abc	188	268	-0.031	0.797	0.412	acde	383	1076	0.200	0.889	0.263
ce	140	498	0.316	0.718	0.219	abcd	188	268	-0.031	0.797	0.412	abde	383	1084	0.195	0.889	0.261
be	149	632	0.254	0.764	0.191	acde	188	268	-0.031	0.797	0.412	be	337	938	0.182	0.782	0.264
bce	149	662	0.230	0.764	0.184	e	155	244	-0.096	0.657	0.388	bc	269	694	0.180	0.624	0.279
ae	149	698	0.201	0.764	0.176	ce	155	244	-0.096	0.657	0.388	abcde	383	1114	0.176	0.889	0.256
abe	149	698	0.201	0.764	0.176	ae	155	244	-0.096	0.657	0.388	ab	269	706	0.172	0.624	0.276
ace	149	728	0.177	0.764	0.170	ace	155	244	-0.096	0.657	0.388	bce	337	968	0.163	0.782	0.258
abce	149	728	0.177	0.764	0.170	be	188	306	-0.148	0.797	0.381	abe	337	1004	0.140	0.782	0.251
b	81	372	0.115	0.415	0.179	abe	188	306	-0.148	0.797	0.381	abc	269	760	0.138	0.624	0.261
bc	81	426	0.072	0.415	0.160	bce	188	306	-0.148	0.797	0.381	abce	337	1034	0.120	0.782	0.246
a	81	438	0.062	0.415	0.156	bde	188	306	-0.148	0.797	0.381	ae	304	942	0.103	0.705	0.244
ab	81	438	0.062	0.415	0.156	abde	188	306	-0.148	0.797	0.381	ace	304	972	0.083	0.705	0.238
c	36	188	0.033	0.185	0.161	bcde	188	306	-0.148	0.797	0.381	c	36	188	-0.037	0.084	0.161
ac	81	492	0.018	0.415	0.141	abce	188	306	-0.148	0.797	0.381	a	81	438	-0.092	0.188	0.156
abc	81	492	0.018	0.415	0.141	abcde	188	306	-0.148	0.797	-	ac	81	492	-0.127	0.188	0.141

MM - Main Metric. TM - Tiebreaker Metric. Rec. - Recall. Prec. - Precision.

Infor. - Informedness. F-Meas. - F-Measure. Marked. - Markedness.

Tools: a - phpSAFE. b - RIPS. c - WAP. d - Pixy. e - WeVerca.

Table E.7.: Best Solutions for the synthetic dataset: XSS and SQLi + XSS: Medium-quality

SQLi						XSS						SQLi + XSS					
Tools	TP	FP	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R	Tools	TP	FP	MM	TM	P/R
Medium-quality			F-Meas.	Rec.	Prec.	F-Meas.			Rec.	Prec.	F-Meas.			Rec.	Prec.		
e	132	882	0.205	0.478	0.130	d	81	90	0.626	0.921	0.474	ce	176	910	0.243	0.484	0.162
ce	132	882	0.205	0.478	0.130	ad	81	90	0.626	0.921	0.474	e	176	910	0.243	0.484	0.162
ae	132	948	0.195	0.478	0.122	bd	81	90	0.626	0.921	0.474	ace	176	976	0.232	0.484	0.153
ace	132	948	0.195	0.478	0.122	cd	81	90	0.626	0.921	0.474	ae	176	976	0.232	0.484	0.153
de	265	2588	0.169	0.960	0.093	de	81	90	0.626	0.921	0.474	cde	346	2678	0.204	0.951	0.114
cde	265	2588	0.169	0.960	0.093	cde	81	90	0.626	0.921	0.474	de	346	2678	0.204	0.951	0.114
be	167	1553	0.167	0.605	0.097	bde	81	90	0.626	0.921	0.474	acde	346	2744	0.200	0.951	0.112
bce	167	1553	0.167	0.605	0.097	ade	81	90	0.626	0.921	0.474	ade	346	2744	0.200	0.951	0.112
ade	265	2654	0.166	0.960	0.091	abd	81	90	0.626	0.921	0.474	d	306	2400	0.199	0.841	0.113
acde	265	2654	0.166	0.960	0.091	acd	81	90	0.626	0.921	0.474	cd	306	2400	0.199	0.841	0.113
abe	167	1619	0.162	0.605	0.094	bcd	81	90	0.626	0.921	0.474	bcd	345	2811	0.196	0.948	0.109
abce	167	1619	0.162	0.605	0.094	abcd	81	90	0.626	0.921	0.474	bd	345	2811	0.196	0.948	0.109
bd	264	2721	0.162	0.957	0.088	abde	81	90	0.626	0.921	0.474	bce	211	1581	0.196	0.580	0.118
bcd	264	2721	0.162	0.957	0.088	bcde	81	90	0.626	0.921	0.474	be	211	1581	0.196	0.580	0.118
bde	274	2837	0.162	0.993	0.088	acde	81	90	0.626	0.921	0.474	ad	306	2466	0.195	0.841	0.110
bcde	274	2837	0.162	0.993	0.088	abcde	81	90	0.626	0.921	0.474	acd	306	2466	0.195	0.841	0.110
d	225	2310	0.160	0.815	0.089	e	44	28	0.550	0.500	0.611	bde	355	2927	0.195	0.975	0.108
cd	225	2310	0.160	0.815	0.089	ce	44	28	0.550	0.500	0.611	bcde	355	2927	0.195	0.975	0.108
abde	274	2903	0.159	0.993	0.086	be	44	28	0.550	0.500	0.611	abcd	345	2877	0.192	0.948	0.107
abcde	274	2903	0.159	0.993	0.086	ae	44	28	0.550	0.500	0.611	abd	345	2877	0.192	0.948	0.107
abd	264	2787	0.159	0.957	0.087	abe	44	28	0.550	0.500	0.611	abde	355	2993	0.191	0.975	0.106
abcd	264	2787	0.159	0.957	0.087	bce	44	28	0.550	0.500	0.611	abcde	355	2993	0.191	0.975	0.106
ad	225	2376	0.156	0.815	0.087	ace	44	28	0.550	0.500	0.611	abce	211	1647	0.190	0.580	0.114
acd	225	2376	0.156	0.815	0.087	abce	44	28	0.550	0.500	0.611	abe	211	1647	0.190	0.580	0.114
b	105	1155	0.137	0.380	0.083	a	0	0	-	0.000	-	b	105	1155	0.129	0.288	0.083
bc	105	1155	0.137	0.380	0.083	b	0	0	-	0.000	-	bc	105	1155	0.129	0.288	0.083
ab	105	1221	0.131	0.380	0.079	c	0	0	-	0.000	-	ab	105	1221	0.124	0.288	0.079
abc	105	1221	0.131	0.380	0.079	ab	0	0	-	0.000	-	abc	105	1221	0.124	0.288	0.079
c	0	0	-	-	-	ac	0	0	-	0.000	-	ac	0	66	-	0.000	0.000
a	0	66	-	0.000	0.000	bc	0	0	-	0.000	-	a	0	66	-	0.000	0.000
ac	0	66	-	0.000	0.000	abc	0	0	-	0.000	-	c	0	0	-	-	-

MM - Main Metric. TM - Tiebreaker Metric. Rec. - Recall. Prec. - Precision.

Infor. - Informedness. F-Meas. - F-Measure. Marked. - Markedness.

Tools: a - phpSAFE. b - RIPS. c - WAP. d - Pixy. e - WeVerca.

Table E.8.: Best Solutions for the synthetic dataset regardless scenarios: XSS and SQLi + XSS

SQLi					XSS					SQLi + XSS						
Tools	TP	FP	Pg	MM	TM	Tools	TP	FP	Pg	MM	TM	Tools	TP	FP	MM	TM
Rec. Prec.					Rec. Prec.					Rec. Prec.						
bde	824	5621		0.253	0.128	abde	3559	4588		0.017	0.437	d	812	1836	0.026	0.154
bcde	824	5711		0.243	0.126	abcde	3559	4628		0.010	0.435	de	1612	4264	0.009	0.306
abde	824	5753		0.238	0.125	ade	3555	4500		0.031	0.441	bd	1714	4724	-0.003	0.326
abcde	824	5843		0.227	0.124	acde	3555	4540		0.024	0.439	bcd	3460	8135	0.091	0.657
ade	802	5261		0.270	0.132	bde	3495	4572		0.005	0.433	cd	4383	10471	0.104	0.833
acde	802	5351		0.260	0.130	bcde	3495	4612		-0.002	0.431	abd	3237	7876	0.067	0.615
de	797	4693		0.331	0.145	de	3479	4308		0.047	0.447	cde	3418	7735	0.111	0.649
cde	797	5002		0.295	0.137	cde	3479	4469		0.019	0.438	abcd	4383	10341	0.113	0.833
bd	666	4575		0.201	0.127	abcd	2794	3266		0.072	0.461	ad	3237	7744	0.076	0.615
abd	666	4707		0.185	0.124	abd	2752	3028		0.104	0.476	e	914	2296	0.014	0.174
bcd	666	4737		0.182	0.123	bcd	2730	3250		0.060	0.457	acd	3050	6793	0.107	0.579
abcd	666	4869		0.167	0.120	bd	2688	3012		0.092	0.472	bde	4357	9891	0.139	0.828
be	578	3820		0.192	0.131	abe	2659	3792		-0.051	0.412	ade	3146	6848	0.121	0.598
bce	578	3910		0.181	0.129	abce	2659	3834		-0.058	0.410	bcde	3008	6393	0.126	0.571
abe	578	3952		0.176	0.128	ae	2616	3680		-0.041	0.416	b	4357	9761	0.148	0.828
abce	578	4042		0.166	0.125	ace	2616	3722		-0.049	0.413	ce	3146	6716	0.130	0.598
ad	576	3885		0.182	0.129	be	2595	3776		-0.063	0.407	acde	1548	4116	0.008	0.294
acd	576	4047		0.163	0.125	bce	2595	3818		-0.070	0.405	abde	1650	4576	-0.005	0.313
d	531	3141		0.219	0.145	e	2535	3481		-0.025	0.421	be	3396	7987	0.089	0.645
cd	531	3582		0.168	0.129	ce	2535	3650		-0.055	0.410	bc	4319	10323	0.102	0.820
ae	530	3036		0.230	0.149	acd	2474	2746		0.089	0.474	abcde	3173	7728	0.065	0.603
ace	530	3126		0.219	0.145	ad	2432	2508		0.121	0.492	ab	3354	7587	0.109	0.637
ce	521	2752		0.253	0.159	cd	2232	2428		0.089	0.479	bce	4319	10193	0.111	0.820
e	494	2344		0.270	0.174	d	2160	1980		0.151	0.522	abe	3173	7596	0.074	0.603
b	312	2412		0.063	0.115	abc	1402	2018		-0.030	0.410	abc	507	1330	0.004	0.096
ab	312	2544		0.048	0.109	bc	1338	2002		-0.042	0.401	abce	2763	6010	0.107	0.525
bc	312	2574		0.044	0.108	ab	1300	1720		-0.002	0.431	ae	4276	9471	0.153	0.812
abc	312	2706		0.029	0.103	b	1236	1704		-0.013	0.420	ace	3056	6402	0.135	0.581
a	156	972		0.059	0.138	ac	758	1162		-0.029	0.395	c	2691	5121	0.155	0.511
ac	156	1134		0.040	0.121	a	656	864		0.000	0.432	a	4276	9001	0.186	0.812
c	99	618		0.037	0.138	c	408	712		0.094	0.364	ac	3029	5825	0.170	0.575

MM - Main Metric. TM - Tiebreaker Metric. Rec. - Recall. Prec. - Precision.

Infor. - Informedness. F-Meas. - F-Measure. Marked. - Markedness.

Tools: a - phpSAFE. b - RIPS. c - WAP. d - Pixy. e - WeVerca.

List of Abbreviations and Symbols

AJAX	Asynchronous Javascript and XML.....	15
API	Application Programming Interface	19
AST	Abstract Syntax Tree	32
BAS	Benchmark Accuracy Score	52
BO	Buffer Overflow	13
BSA	Benchmark for Security Automation.....	6
Bugtraq	Electronic mailing list dedicated to issues about computer security	76
CAS	Center for Assured Software.....	41
CBO	Coupling Between Objects	84
CCN	Cyclomatic Complexity Number.....	71
CCN2	Extended Cyclomatic Complexity Number	77
CFG	Control-Flow Graph.....	31
DFG	Data-Flow Graph	31
CIA	Confidentiality, Integrity and Availability.....	10
CIS	Number of non-private methods and properties.....	84
CMS	Content Management System.....	i
CSS	Cascading Style Sheets.....	15
CSV	Comma Separated Values	58
CVE	Common Vulnerability Enumeration	50
CVSS	Common Vulnerability Scoring System.....	164
CWE	Common Weakness Enumeration.....	5
DA	Dynamic Analysis.....	ii
DAST	Dynamic Application Security Testing.....	48
DBMS	Database Management System.....	29
DF	Density Function	
DLD	Duplicated Line Density	84
DOM	Document Object Model.....	19
DR	Discrimination Rate.....	41

EP	Entry Point	18
FN	False Negative	ii
FP	False Positive	ii
FPR	False Positive Rate	52
HA	Hybrid Analysis	16
HTML	HyperText Markup Language	2
HTTP	Hypertext Transfer Protocol	2
HTTPS	Hypertext Transfer Protocol Secure	16
IDE	Integrated Development Environment	
IDS	Intrusion Detection System	2
IP	Internet Protocol	3
IPS	Intrusion Prevention Systems	2
LDAP	Lightweight Directory Access Protocol	18
LFI	Local File Inclusion	38
LLOC	Logical Lines of Code	84
LOC	Line of Code	63
MFE	Malicious File Execution	36
ML	Machine Learning	163
N	Negative Instances	74
NIST	National Institute of Standards and Technology	10
NPARAM	Number of parameters in functions and methods	84
NPATH	Number of execution paths	84
NSA	National Security Agency	41
NVLOC	Non-Vulnerable Line of Code	79
NVLOCs	Non-Vulnerable Lines of Code	80
OOP	Object-Oriented Programming	i
ORM	Object Relational Mapper	18
OS	Operating System	59
OSC	OS Command	18
OSCI	OS Command Injection	38
OWASP	Open Web Application Security Project	4
P	Positive Instances	74
PT	Path Traversal	32
PE	Privilege Escalation	83
PHP	PHP Hypertext Preprocessor	14
PoC	Proof of Concept	76
POP	Procedure Oriented Programming	84
PQL	Program Query Language	35
PT	Parse Tree	32
QMOOD	Quality Model for Object Oriented Design	167
RCE	Remote Code Execution	37
RFI	Remote File Inclusion	38
ROC	Receiver Operating Characteristic	107
SA	Static Analysis	ii
SAMATE	Software Assurance Metrics and Tool Evaluation	6

SANS	System Administration, Networking and Security	5
SARD	Software Assurance Reference Dataset	41
SAST	Static Application Security Testing	i
SATE	Static Analysis Tool Exposition	52
SCM	Source Code Metric	78
SD	Synthetic Dataset	198
SDLC	Software Development Life Cycle	3
SF	Sanitization Function	86
SIG	Software Improvement Group	173
SM	Security Misconfiguration	5
SMT	Satisfiability Modulo Theories	37
SMTP	Simple Mail Transfer Protocol	18
SPP	Software Product Property	78
SPPs	Software Product Properties	167
SQALE	Software Quality Assessment based on Lifecycle Expectations	167
SQL	Structured Query Language	5
SQLi	SQL Injection	i
SQLIA	SQL Injection Attack	26
SQuaRE	Systems and Software Quality Requirements and Evaluation	167
SS	Sensitive Sink	32
SSL	Secure Sockets Layer	2
SwMM-RSV	Software Measures and Metrics to Reduce Security Vulnerabilities	166
TLS	Transport Layer Security	2
TN	True Negative	52
TNR	True Negative Rate	53
TP	True Positive	40
TPR	True Positive Rate	53
UFU	Unrestricted File Upload	13
UI	User Interface	24
URL	Uniform Resource Locator	19
VLOC	Vulnerable Line of Code	79
VLOCs	Vulnerable Lines of Code	79
WAF	Web Application Firewall	2
WMC	Weighted Method Count	84
WS	Web Service	40
WPD	WordPress Plugins Dataset	83
WPPD	WordPress Plugin Directory	4
WPVD	WordPress Vulnerability Database	76
WWW	World Wide Web	2
XML	Extensible Markup Language	13
XML	eXtensible Markup Language	13
XPath	XML Path Language	13
XSS	Cross-Site Scripting	i

Bibliography

- [1] Vadim Okun, William F Guthrie, Romain Gaucher, and Paul E Black. Effect of static analysis tools on software security: preliminary investigation. In *Proceedings of the 2007 ACM workshop on Quality of protection*, pages 1–5. ACM, 2007. [cited at page 2, 40, 98]
- [2] H. Atashzar, A. Torkaman, M. Bahrololum, and M.H. Tadayon. A survey on web application vulnerabilities and countermeasures. In *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pages 647–652, Nov 2011. [cited at page 2]
- [3] OWASP - Testing Guide 4.0. https://www.owasp.org/images/5/52/OWASP_Testing_Guide_v4.pdf, April 2015. Last accessed 1 March 2015. [cited at page 2]
- [4] Herjavec Group. 2019 Official Annual Cybercrime Report. *Cybersecurity Ventures sponsored by Herjavec Group*, page 12, 2019. [cited at page 2]
- [5] datareportal. <https://datareportal.com/global-digital-overview>. Last accessed 2 November 2021. [cited at page 3]
- [6] Steve Morgan. 2021 REPORT: CYBERWARFARE IN THE C-SUITE, Jan 2021. Last accessed 15 May 2021. [cited at page 3, 10]
- [7] Steve Morgan. 2019 Official Annual Cybercrime Report, 2019. Last accessed 15 June 2019. [cited at page 3]
- [8] statista.com. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. <https://www.statista.com/statistics/871513/worldwide-data-created>, 2021. Last accessed 7 June 2021. [cited at page 3]
- [9] statista.com. Iot and non-iot connections worldwide 2010-2025. <https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide>, 2021. Last accessed 7 June 2021. [cited at page 3]

-
- [10] deloitte.com. Wearable technology in health care: Getting better all the time. Last accessed 10 December 2021. [cited at page 3]
- [11] Sajjad Rafique, Mamoonah Humayun, Zartasha Gul, Ansar Abbas, and Hasan Javed. Systematic review of web application security vulnerabilities detection methods. *Journal of Computer and Communications*, 03:28–40, 01 2015. [cited at page 3]
- [12] Keith Ward. 2013 Microsoft Security Study. <https://visualstudiomagazine.com/articles/2013/07/16/majority-of-us-devs-dont-practice-secure-coding.aspx>, 2013. Last accessed 7 April 2020. [cited at page 3]
- [13] National Cyber Security Centre. Secure development and deployment guidance. <https://www.ncsc.gov.uk/collection/developers-collection/principles/keep-your-security-knowledge-sharp>, 2019. Last accessed 20 February 2019. [cited at page 3]
- [14] S. Rafique, M. Humayun, B. Hamid, A. Abbas, M. Akhtar, and K. Iqbal. Web application security vulnerabilities detection approaches: A systematic mapping study. In *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6, June 2015. [cited at page 4]
- [15] A. van der Stock, B. Glas, N. Smithline, and T. Gigler. OWASP Top 10 - 2017: The ten most critical web application security risks. Technical report, OWASP Foundation, 2017. [cited at page 4, 13, 18, 83]
- [16] WhiteHat Security. Application Security Statistics Report: The Case for DevSecOps, 2017. Last accessed 15 March 2018. [cited at page 4]
- [17] Tom Smith. The most important elements of application security. <https://dzone.com/articles/the-most-important-elements-of-application-security>, 2015. Last accessed 6 November 2015. [cited at page 4]
- [18] Jaepyo Park, Yeunsoo Choo, and Jonghee Lee. A Hybrid Vulnerability Analysis Tool Using a Risk Evaluation Technique. *Wireless Personal Communications*, 105(2):443–459, mar 2019. [cited at page 4, 48]
- [19] Website hacked trend report 2016-Q1. <https://sucuri.net/website-security/Reports/Sucuri-Website-Hacked-Report-2016Q1.pdf>, 2016. [cited at page 4]
- [20] https://w3techs.com/technologies/overview/content_management/all, 2021. Last accessed 17 December 2021. [cited at page 4]
- [21] BuiltWith. <https://trends.builtwith.com/cms>, 2021. Last accessed 19 December 2021. [cited at page 4]
- [22] https://w3techs.com/technologies/overview/programming_language, 2020. Last accessed 17 December 2021. [cited at page 4, 133]
- [23] Huda Khan, Deven Shah, and A Application Security Risk. Webapps security with rips. (December):107–112, 2012. [cited at page 5]

- [24] 2019 website threat research report. <https://sucuri.net/wp-content/uploads/2020/01/20-sucuri-2019-hacked-report-1.pdf>, 2020. [cited at page 5]
- [25] Jason Bau, Frank Wang, Elie Bursztein, Patrick Mutchler, and John C Mitchell. Vulnerability factors in new web applications: Audit tools, developer selection & languages. 2013. [cited at page 5]
- [26] Atul S Choudhary and ML Dhore. Cidit: Detection of malicious code injection attacks on web application. *International Journal Of Computing Applications*, 52(0):19–25, 2012. [cited at page 5]
- [27] OWASP Foundation. OWASP Top 10 - 2021. <https://owasp.org/Top10/>, 2021. Last accessed 5 October 2021. [cited at page 5, 17]
- [28] 2014 global report on the cost of cyber crime, <http://www.ponemon.org/library/2014-global-report-on-the-cost-of-cyber-crime>. <http://www.ponemon.org/library/2014-global-report-on-the-cost-of-cyber-crime>, October 2014. Last accessed 15 October 2015. [cited at page 9]
- [29] A. Buecker, S. Arunkumar, B. Blackshaw, M. Borrett, P. Brittenham, J. Flegr, J. Jacobs, V. Jeremic, M. Johnston, C. Mark, et al. *Using the IBM Security Framework and IBM Security Blueprint to Realize Business-Driven Security*. IBM redbooks. IBM Redbooks, 2014. [cited at page 10]
- [30] Computer security concepts. <http://hitachi-id.com/concepts>, 2015. Last accessed 25 July 2015. [cited at page 10]
- [31] Patrick D. Gallagher. Glossary of key information security terms - revision 2 (nistir 7298), <http://nvlpubs.nist.gov/nistpubs/ir/2013/nist.ir.7298r2.pdf>. <http://nvlpubs.nist.gov/nistpubs/ir/2013/NIST.IR.7298r2.pdf>, May 2013. Last accessed 12 August 2017. [cited at page 10]
- [32] Donn B Parker. Toward a new framework for information security. *FLY*, page 501, 2002. [cited at page 11]
- [33] information-security. <http://www.pimconsultancy.com/uk-data-protection-act/information-security>, 2015. Last accessed 12 July 2015. [cited at page 12]
- [34] Debbie Walkowski. Threats, vulnerabilities, exploits and their relationship to risk. <https://www.f5.com/labs/articles/education/vulnerabilities-threats-exploits-and-their-relationship-to-risk>. Last accessed 14 April 2021. [cited at page 12]
- [35] Joint Task Force Transformation Initiative. Guide for conducting risk assessment. <https://csrc.nist.gov/publications/detail/sp/800-30/rev-1/final>, 2012. Last accessed 16 September 2020. [cited at page 12]
- [36] https://csrc.nist.gov/glossary/term/threat_agent_source. Last accessed 31 January 2021. [cited at page 13]

-
- [37] <https://csrc.nist.gov/glossary/term/vulnerability>. Last accessed 31 January 2021. [cited at page 13]
- [38] Mary E. Shacklett. Attack vector. <https://www.techtarget.com/searchsecurity/definition/attack-vector>. Last accessed 30 April 2021. [cited at page 13]
- [39] David Cramer. It security vulnerability vs threat vs risk. <https://www.bmc.com/blogs/security-vulnerability-vs-threat-vs-risk-whats-difference/#>. Last accessed 14 May 2020. [cited at page 13]
- [40] <https://csrc.nist.gov/glossary/term/countermeasures>. Last accessed 14 April 2021. [cited at page 13]
- [41] OWASP Foundation. OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks. Technical report, OWASP Foundation, 2017. [cited at page 14, 17, 29]
- [42] Xiaowei Li and Yuan Xue. A survey on server-side approaches to securing web applications. *ACM Comput. Surv.*, 46(4):54:1–54:29, mar 2014. [cited at page 15, 16, 18]
- [43] <http://www.securityfocus.com>, 2015. Last accessed 13 July 2015. [cited at page 16]
- [44] Gartner, inc. <http://www.gartner.com>, 2015. Last accessed 25 July 2015. [cited at page 16]
- [45] Hasty Atashzar, Atefeh Torkaman, Marjan Bahrololum, and Mohammad H Tadayon. A survey on web application vulnerabilities and countermeasures. *2011 6Th International Conference on Computer Sciences and Convergence Information Technology Iccit*, (Mic):647–652, 2012. [cited at page 16]
- [46] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3):11:1–11:46, jun 2012. [cited at page 16]
- [47] Mukesh Kumar Gupta, MC Govil, and Girdhari Singh. Static analysis approaches to detect sql injection and cross site scripting vulnerabilities in web applications: A survey. In *Recent Advances and Innovations in Engineering (ICRAIE), 2014*, pages 1–5. IEEE, May 2014. [cited at page 16, 18, 34, 38]
- [48] Mehdi Achour. Php manual. <http://php.net/manual>, 2015. Last accessed 16 August 2015. [cited at page 16]
- [49] OWASP Top 10 - 2013 - The Ten Most Critical Web Application Security Risks. https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013, June 2013. Last accessed 12 June 2015. [cited at page 17]
- [50] Stephan Neuhaus and Thomas Zimmermann. Security trend analysis with cve topic models. In *2010 IEEE 21st international symposium on Software reliability engineering (ISSRE)*, pages 111–120. IEEE, 2010. [cited at page 18, 31, 62, 83]

- [51] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *In Symposium on Network and Distributed System Security (NDSS*, number February, pages 23–26, 2014. [cited at page 18, 32, 37, 38, 40, 58, 85, 86, 138, 183]
- [52] Oslien Mesa, Reginaldo Vieira, Marx Viana, Vinicius Durelli, Elder Cirilo, Marcos Kalinowski, and Carlos Lucena. Understanding vulnerabilities in plugin-based web systems: An exploratory study of wordpress. 09 2018. [cited at page 18]
- [53] Wordpress Vulnerability Database. <https://patchstack.com/database>, 2021. Last accessed 8 March 2021. [cited at page 18]
- [54] Common Weakness Enumeration. <http://cwe.mitre.org/top25>, 2020. Last accessed 4 June 2020. [cited at page 18]
- [55] Dennis Odell. *Pro Javascript RIA Techniques: Best Practices, Performance and Presentation*. Apress, 2009. Last accessed 5 July 2016. [cited at page 18]
- [56] Excess-xss.com. <http://excess-xss.com>, 2015. Last accessed 26 July 2015. [cited at page 19, 20, 21, 22, 23]
- [57] Learn more about web site security, <http://www.acunetix.com/websitesecurity>. <http://www.acunetix.com/websitesecurity>, July 2015. Last accessed 15 July 2015. [cited at page 19]
- [58] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. [cited at page 20]
- [59] The php group, <http://www.php.net>. <http://www.php.net>, 2021. Last accessed 9 October 2021. [cited at page 20]
- [60] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 199–209, Washington, DC, USA, 2009. IEEE Computer Society. [cited at page 21]
- [61] S. Fogie, J. Grossman, R. Hansen, a. Rager, and P.D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. 2007. [cited at page 22]
- [62] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. 2007. [cited at page 24, 31, 39]
- [63] Plugin Handbook. <https://developer.wordpress.org/plugins>, 2021. Last accessed 10 July 2021. [cited at page 24]
- [64] XSS (Cross Site Scripting) Prevention Cheat Sheet. [https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet), 2015. Last accessed 10 June 2015. [cited at page 25]

- [65] William G. J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006. [cited at page 25, 26]
- [66] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 989–1003, San Diego, CA, 2014. USENIX Association. [cited at page 26]
- [67] Stephen Thomas, Laurie Williams, and Tao Xie. On automated prepared statement generation to remove sql injection vulnerabilities. *Inf. Softw. Technol.*, 51(3):589–598, Mar 2009. [cited at page 29]
- [68] Pl/sql procedure. <https://www.oracletutorial.com/plsql-tutorial/plsql-procedure>, 2021. Last accessed 5 March 2020. [cited at page 30]
- [69] Aybuke Aurum, Håkan Petersson, and Claes Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, 2002. [cited at page 30]
- [70] Islam Elkhalfa and Bilal Ilyas. *Static Code Analysis: A Systematic Literature Review and an Industrial Survey*. PhD thesis, Faculty of Computing - Blekinge Institute of Technology, SE - 371 79 Karlskrona, Sweden, 9 2016. Thesis no: MSSE-2016-09. [cited at page 30]
- [71] IEEE Standard for Software Reviews and Audits. *IEEE Std 1028-2008*, pages 1–53, 2008. [cited at page 30]
- [72] Rips. <http://rips-scanner.sourceforge.net>, 2020. Last accessed. [cited at page 31]
- [73] Wap. <http://awap.sourceforge.net>, 2020. Last accessed. [cited at page 31]
- [74] phpsafe. <https://github.com/JoseCarlosFonseca/phpSAFE>, 2020. Last accessed. [cited at page 31]
- [75] Veracode white box testing. <http://www.veracode.com>, 2020. Last accessed 5 March 2020. [cited at page 31]
- [76] Cxsast. <https://www.checkmarx.com/products/static-application-security-testing>, 2020. Last accessed 5 March 2020. [cited at page 31]
- [77] Analyzer. <https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer>, 2020. Last accessed 5 March 2020. [cited at page 31]
- [78] Sonarqube. <https://www.sonarqube.org/features/security>, 2020. Last accessed 5 March 2020. [cited at page 31]
- [79] Nexhati Alija. Justification of Software Maintenance Costs. *International Journal of Advanced Research in Computer Science and Software Engineering*, 7(3):1–53, 3 2017. [cited at page 31]

- [80] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. *Proceedings of the 13th conference on World Wide Web - WWW '04*, page 40, 2004. [cited at page 31, 35, 38, 39]
- [81] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy*, pages 6 pp.–263, May 2006. [cited at page 31, 35, 37, 38, 39, 48, 86, 183]
- [82] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE'08*, pages 171–180. IEEE, 2008. [cited at page 31, 36, 39]
- [83] D. Balzarotti, M. Cova, V. Felmetzger, N. Jovanovic, E. Kirda, C. Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy, 2008. SP 2008.*, pages 387–401, May 2008. [cited at page 31, 36, 39, 40, 46, 47, 48, 135]
- [84] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM. [cited at page 31]
- [85] Jiri Slaby. *Automatic Bug-finding Techniques for Large Software Projects*. PhD thesis, PhD thesis. Masaryk University, 2013. [cited at page 34]
- [86] Yunhui Zheng and Xiangyu Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 652–661, May 2013. [cited at page 34, 37, 39]
- [87] Developing secure Web applications: An introduction to IBM Rational AppScan Developer Edition. http://www.ibm.com/developerworks/rational/library/08/0916_podjarny, 2015. Last accessed 9 June 2015. [cited at page 34]
- [88] Karl Chen and David Wagner. Large-scale analysis of format string vulnerabilities in debian linux. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS '07*, pages 75–84, New York, NY, USA, 2007. ACM. [cited at page 34]
- [89] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10, SSYM'01*, Berkeley, CA, USA, 2001. USENIX Association. [cited at page 35]
- [90] Aske Simon Christensen, Anders Moller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Conference on Static Analysis, SAS'03*, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag. [cited at page 35, 39]
- [91] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web, WWW '05*, pages 432–441, New York, NY, USA, 2005. ACM. [cited at page 35, 39]

-
- [92] Inc. Armorize Technologies. codesecure. <http://www.armorize.com>. Last accessed 3 March 2021. [cited at page 35]
- [93] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association. [cited at page 35, 38, 39, 62]
- [94] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association. [cited at page 35, 39]
- [95] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 154–157. Springer, 2010. [cited at page 36, 39]
- [96] Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic string verification: An automata-based approach. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software*, pages 306–324, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. [cited at page 36]
- [97] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 605–609, 2009. [cited at page 36]
- [98] Xin hua Zhang and Zhi jian Wang. Notice of retraction a static analysis tool for detecting web application injection vulnerabilities for asp program. In *2010 2nd International Conference on e-Business and Information System Security (EBISS)*, pages 1–5, May 2010. [cited at page 37, 39]
- [99] David Hauzar and Jan Kofron. On security analysis of php web applications. *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pages 577–582, jul 2012. [cited at page 37, 39]
- [100] David Hauzar and Jan Kofron. Framework for Static Analysis of PHP Applications. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 689–711, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. [cited at page 37, 39, 86, 183]
- [101] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *Proceedings of the First International Conference on Runtime Verification, RV'10*, page 300–314, Berlin, Heidelberg, 2010. Springer-Verlag. [cited at page 37]
- [102] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007. [cited at page 37, 39]

- [103] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd International Conference on World Wide Web, WWW '14*, page 63–74, New York, NY, USA, 2014. Association for Computing Machinery. [cited at page 38, 39, 40, 46, 86, 136, 183]
- [104] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C. Briand. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 642–651, Piscataway, NJ, USA, 2013. IEEE Press. [cited at page 38]
- [105] Sreenivasa Rao and N Kumar. Web application vulnerability detection using dynamic analysis with penetration testing. *International Journal of Computer Science and Security*, 6(2):1, 2012. [cited at page 38, 48]
- [106] Perl security. <http://perldoc.perl.org/perlsec.html>, 2015. Last accessed 9 June 2015. [cited at page 38, 43, 95]
- [107] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *Computer Security Applications Conference, 21st Annual*, pages 9 pp.–311, Dec 2005. [cited at page 38, 43]
- [108] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992. [cited at page 40]
- [109] Paulo Nunes, José Fonseca, and Marco Vieira. phpSAFE: A security analysis tool for OOP web application plugins. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 299–306, June 2015. [cited at page 40, 85, 86, 183]
- [110] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering, ISSRE '04*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society. [cited at page 40, 98]
- [111] N. Meng, Q. Wang, Q. Wu, and H. Mei. An approach to merge results of multiple static analysis tools (short paper). In *2008 The Eighth International Conference on Quality Software*, pages 169–174, Aug 2008. [cited at page 40, 98]
- [112] Q. Wang, N. Meng, Z. Zhou, J. Li, and H. Mei. Towards soa-based code defect analysis. In *IEEE International Symposium on Service-Oriented System Engineering, 2008. SOSE '08*, pages 269–274, Dec 2008. [cited at page 40, 98]
- [113] Fort George Meade. [https://samate.nist.gov/docs/CAS%202012%20Static %20Analysis%20Tool%20Study%20Methodology.pdf](https://samate.nist.gov/docs/CAS%202012%20Static%20Analysis%20Tool%20Study%20Methodology.pdf). [cited at page 41, 52]
- [114] Test Suites. <http://samate.nist.gov/SRD/testsuite.php>, 2018. Last accessed 5 July 2018. [cited at page 41, 48, 195]

-
- [115] On Analyzing Static Analysis Tools. https://media.blackhat.com/bh-us-11/Willis/BH_US_11_WillisBritton_Analyzing_Static_Analysis_Tools_WP.pdf, 2011. Last accessed 20 June 2021. [cited at page 41]
- [116] Gabriel Díaz and Juan Ramón Bermejo. Static analysis of source code security: Assessment of tools against SAMATE tests. *Information and Software Technology*, 55(8):1462–1476, aug 2013. [cited at page 41, 98]
- [117] SAMATE - Software Assurance Metrics And Tool Evaluation. <http://samate.nist.gov>, June 2015-06-12. Last accessed 12 June 2015. [cited at page 41, 52, 70, 72, 89, 98, 204]
- [118] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 1, pages 470–481, Mar 2016. [cited at page 41, 98]
- [119] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *35th International Conference on Software Engineering*, pages 672–681. IEEE, may 2013. [cited at page 41, 98, 138]
- [120] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *SIGSOFT Softw. Eng. Notes*, 22(6):432–449, nov 1997. [cited at page 41, 42]
- [121] Thoms Bell. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234, 1999. [cited at page 42]
- [122] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli. Detecting control flow in smartphones: Combining static and dynamic analyses. volume 7672, pages 33–47, 12 2012. [cited at page 42]
- [123] J E Jr Kimble and L J White. An alternative source code analysis. *2000 Proceedings International Conference on Software Maintenance*, 10(2):64–75, 2000. [cited at page 42, 45]
- [124] Mamdouh Alenezi, Muhammad Nadeem, and Raja Asif. Sql injection attacks countermeasures assessments. *Indonesian Journal of Electrical Engineering and Computer Science*, 21, 10 2020. [cited at page 42]
- [125] Min Gyung, Kang Stephen, Mccamant Pongsin, and Poosankam Dawn. Dta ++ : Dynamic taint analysis with targeted control-flow propagation. *Work*, 2011. [cited at page 42]
- [126] Ruby security feature, <https://ruby-hacking-guide.github.io/security.html>. <https://ruby-hacking-guide.github.io/security.html>, 2015-06-09. Last accessed 9 June 2015. [cited at page 43]
- [127] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. *Automatically hardening web applications using precise tainting*. Springer, 2005. [cited at page 43]
-

- [128] Brian Chess and Jacob West. Dynamic taint propagation: Finding vulnerabilities without attacking. *Information Security Technical Report*, 13(1):33–39, 2008. [cited at page 43]
- [129] William G. J. Halfond and Alessandro Orso. Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 174–183, New York, NY, USA, 2005. ACM. [cited at page 43, 45, 135]
- [130] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: Preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 12–24, New York, NY, USA, 2007. ACM. [cited at page 43]
- [131] Muhammad Nouman, Usman Pervez, Osman Hasan, and Kashif Saghar. Software testing: A survey and tutorial on white and black-box testing of c/c++ programs. In *2016 IEEE Region 10 Symposium (TENSymp)*, pages 225–230, 2016. [cited at page 44]
- [132] Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira. An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal*, 2009. [cited at page 44]
- [133] Adam Doupe, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'10, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag. [cited at page 44]
- [134] José Fonseca, Marco Vieira, and Henrique Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, PRDC '07, pages 365–372, Washington, DC, USA, 2007. IEEE Computer Society. [cited at page 44]
- [135] Mohd Ehmer Khan et al. Different approaches to white box testing technique for finding errors. *International Journal of Software Engineering and Its Applications*, 5(3):1–14, 2011. [cited at page 44]
- [136] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. *Proceedings - International Conference on Software Engineering*, pages 199–209, 2009. [cited at page 45]
- [137] Angelo Ciampa, Corrado Aaron Visaggio, and Massimiliano Di Penta. A heuristic-based approach for detecting sql-injection vulnerabilities in web applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 43–49, New York, NY, USA, 2010. ACM. [cited at page 45, 46, 135]
- [138] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 372–382, New York, NY, USA, 2006. ACM. [cited at page 45, 46, 135]

-
- [139] Inyong Lee, Soonki Jeong, Sangsoo Yeo, and Jongsub Moon. A novel method for SQL injection attack detection based on removing SQL query attribute values. *Mathematical and Computer Modelling*, 55(1-2):58–68, 2012. [cited at page 45, 46, 47, 135]
- [140] How correlation (hybrid analysis) works. https://help.hcltechsw.com/appscan/Enterprise/9.0.3/topics/c_how_correlation_works.html, 2020. Last accessed 19 March 2020. [cited at page 46]
- [141] Anita D’Amico. SAST vs DAST: What is the right choice for application security testing? <https://codedx.com/blog/sast-vs-dast-tools>, 2019. Last accessed 25 May 2019. [cited at page 45]
- [142] Sanjay Rawat, Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet. Combining static and dynamic analysis for vulnerability detection. *CoRR*, abs/1305.3883, 2013. [cited at page 46, 135]
- [143] Varun Gupta, Punjab State, and Electricity Board. Measurement of dynamic metrics using dynamic analysis of programs. In *Proceedings of the WSEAS International Conference on Applied Computing Conference*, pages 81–86, 2008. [cited at page 46]
- [144] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 49–59, New York, NY, USA, 2014. Association for Computing Machinery. [cited at page 47]
- [145] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V.N. Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, Baltimore, MD, August 2018. USENIX Association. [cited at page 47]
- [146] Yannis Smaragdakis and Christoph Csallner. Combining static and dynamic reasoning for bug detection. In *Proc. International Conference on Tests And Proofs (TAP)*, volume 4454 of *LNCS*, pages 1–16. Springer, feb 2007. [cited at page 48]
- [147] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):8:1–8:37, may 2008. [cited at page 48]
- [148] Mattia Monga, Roberto Paleari, and Emanuele Passerini. A hybrid analysis framework for detecting web application vulnerabilities. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 25–32. IEEE Computer Society, 2009. [cited at page 48]
- [149] J. Fonseca, M. Vieira, and H. Madeira. Evaluation of web security mechanisms using vulnerability & attack injection. *IEEE Transactions on Dependable and Secure Computing*, 11(5):440–453, Sept 2014. [cited at page 48, 135]
- [150] Xincheng He, L. Xu, and Chunliu Cha. Malicious javascript code detection based on hybrid analysis. *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 365–374, 2018. [cited at page 48]

- [151] Carrie Ballinger. TPC-D: Benchmarking for Decision Support. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993. [cited at page 49, 70, 95]
- [152] Nico L de Poel, Frank B Brokken, and Gerard R Renardel de Lavalette. Automated security review of php web applications with static code analysis. *Master's thesis*, 5, 2010. [cited at page 49, 70]
- [153] Yves Crouzet and Karama Kanoun. System Dependability: Characterization and Benchmarking. In S.Sedigh A.Hurson, editor, *Advances in Computers. Special issue: Dependable and Secure Systems Engineering*, pages 93–139. Elsevier, May 2012. [cited at page 49]
- [154] Lukas M Weber, Wouter Saelens, Robrecht Cannoodt, Charlotte Soneson, Alexander Hapfelmeier, Paul P Gardner, Anne-Laure Boulesteix, Yvan Saeys, and Mark D Robinson. Essential guidelines for computational method benchmarking. *Genome biology*, 20(1):125, 2019. [cited at page 49, 97]
- [155] Marco Vieira, Henrique Madeira, Kai Sachs, and Samuel Kounev. Resilience benchmarking. *Resilience Assessment and Evaluation of Computing Systems*, pages 283–301, 2012. [cited at page 49, 50]
- [156] Sarah Heckman and Laurie Williams. On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques. *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 41–50, 2008. [cited at page 49, 95]
- [157] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005. [cited at page 49]
- [158] Aurelien Delaitre, Bertrand Stivalet, Elizabeth Fong, and Vadim Okun. Evaluating bug finders: Test and measurement of static code analyzers. In *Proceedings of the First International Workshop on Complex faults and Failures in Large Software Systems*, COUFLESS '15, pages 14–20, Florence, Italy, 2015. IEEE Press. [cited at page 49, 50, 52, 70, 95]
- [159] Nuno Antunes and Marco Vieira. On the metrics for benchmarking vulnerability detection tools. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 505–516, June 2015. [cited at page 49, 71, 72, 73, 94]
- [160] Vineeth Kashyap, Jason Ruchti, Lucja Kot, Emma Turetsky, Rebecca Swords, Shih An Pan, Julien Henry, David Melski, and Eric Schulte. Automated customized bug-benchmark generation. *Proceedings - 19th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2019*, pages 103–114, 2019. [cited at page 49]
- [161] James A Kupsch and Barton P Miller. Manual vs. automated vulnerability assessment: A case study. In *The 1st International Workshop on Managing Insider Security Threats (MIST 2009)*, 2009. [cited at page 50]

-
- [162] Peng Li and Baojiang Cui. A comparative study on software vulnerability static analysis techniques and tools. In *2010 IEEE International Conference on Information Theory and Information Security*, pages 521–524, 2010. [cited at page 51]
- [163] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 02 2008. [cited at page 51]
- [164] Martin Johns and Moritz Jodeit. Scanstud: A methodology for systematic, fine-grained evaluation of static analysis tools. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 523–530, 2011. [cited at page 51, 72]
- [165] I. Pashchenko, S. Dashevskiy, and F. Massacci. Delta-bench: Differential benchmark for static analysis security testing tools. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 163–168, 2017. [cited at page 51, 72]
- [166] Juan-Ramón Higuera, Javier Bermejo, Juan Antonio Montalvo, Javier Villalba, and Juan Perez. Benchmarking approach to compare web applications static analysis tools detecting owasp top ten security vulnerabilities. *Computers, Materials and Continua*, 64:1555–1577, 06 2020. [cited at page 51, 72]
- [167] <https://owasp.org/www-project-benchmark>, 2021. Last accessed 15 August 2021. [cited at page 52, 70, 72, 89, 204]
- [168] Paul E. Black, Michael Kass, Michael Koo, and Elizabeth Fong. Source code security analysis tool functional specification version 1.1. (February), 2011. [cited at page 52]
- [169] NIST SOFTWARE QUALITY GROUP. Static analysis tool exposition (sate). <https://www.nist.gov/itl/ssd/software-quality-group/samate/static-analysis-tool-exposition-sate>, 2021. Last accessed 16 August 2021. [cited at page 52]
- [170] José Carlos Coelho Martins da Fonseca and Marco Paulo Amorim Vieira. A practical experience on the impact of plugins in web security. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS)*, pages 21–30. IEEE, 2014. [cited at page 56, 57, 62, 67]
- [171] Automattic. <http://automattic.com>, 2021. Last accessed 9 October 2021. [cited at page 56, 62]
- [172] Usage of content management systems for websites. http://w3techs.com/technologies/overview/content_management/all, November 2014. Last accessed 10 November 2014. [cited at page 62]
- [173] J. Fonseca and M. Vieira. *A Survey on Secure Software Development Lifecycles*. Khalid Buragga, Noor Zaman (Eds.), 2013. [cited at page 65]
- [174] J. Fonseca and M. Vieira. Mapping software faults with web security vulnerabilities. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008.*, pages 257–266, June 2008. [cited at page 66]

- [175] Jonathan Greig. Average time to fix critical cybersecurity vulnerabilities is 205 days: report. <https://cacm.acm.org/news/253585-average-time-to-fix-critical-cybersecurity-vulnerabilities-is-205-days-report/fulltext>, 2021. Last accessed 22 July 2021. [cited at page 67]
- [176] Jóakim von Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. How to build a benchmark. In *ICPE '15*, 2015. [cited at page 71]
- [177] H. M. Kienle and S. E. Sim. Towards a benchmark for web site extractors: a call for community participation. In *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.*, pages 82–87, March 2003. [cited at page 72]
- [178] Aurelien Delaitre, Vadim Okun, and Elizabeth Fong. Of Massive Static Analysis Data. In *Proceedings of the 2013 IEEE Seventh International Conference on Software Security and Reliability Companion, SERE-C '13*, page 163–167, USA, 2013. IEEE Computer Society. [cited at page 72]
- [179] David MW Powers. Evaluation evaluation a Monte Carlo study. *arXiv preprint arXiv:1504.00854*, 2015. [cited at page 75]
- [180] Paul E Black and Elizabeth N. Fong. Report of the Workshop on Software Measures and Metrics to Reduce Security Vulnerabilities (SwMM-RSV). Technical report, National Institute of Standards and Technology, Gaithersburg, MD, nov 2016. [cited at page 76, 95, 166]
- [181] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307, 2012. [cited at page 77, 78, 166, 169, 176]
- [182] pdepend.org. <https://pdepend.org>, 2016. Last accessed 3 November 2016. [cited at page 77, 78, 84, 166]
- [183] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39, Sept 2007. [cited at page 78]
- [184] Wei Hu, Tino Loeffler, and Joachim Wegener. Quality model based on ISO/IEC 9126 for internal quality of MATLAB/Simulink/Stateflow models. In *Industrial Technology (ICIT), 2012 IEEE International Conference on Industrial Technology*, pages 325–330. IEEE, 2012. [cited at page 78]
- [185] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. [cited at page 78]
- [186] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13(2):115–139, June 1974. [cited at page 78]
- [187] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, Jan 2002. [cited at page 78, 167]

-
- [188] Software improvement group (sig). <https://www.sig.eu>, January 2017. Last accessed 5 January 2017. [cited at page 78, 166, 169, 173, 176]
- [189] Irena Bojanova, Paul E. Black, Yaacov Yesha, and Yan Wu. The bugs framework (BF): A structured approach to express bugs. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Vienna, Austria, August 1-3, 2016*, pages 175–182, 2016. [cited at page 79]
- [190] Agnes Talalaev. WORDPRESS SECURITY-WordPress Vulnerability News. <https://www.webarxsecurity.com/vulnerable-plugins-january-2020>, 2020. Last accessed 15 April 2020. [cited at page 83]
- [191] Wordpress plugin directory. <https://wordpress.org/plugins>, 2021. Last accessed 25 July 2021. [cited at page 83]
- [192] Hosting Tribunal. 35+ WordPress Statistics for the Budding Webmaster [Infographic]. <https://hostingtribunal.com/blog/wordpress-statistics>, 2020. Last accessed 3 March 2020. [cited at page 83]
- [193] WPScan Vulnerability Database. <https://wpsvulndb.com>, 2015. Last accessed 25 October 2015. [cited at page 83, 85]
- [194] Paulo Nunes. <https://github.com/pjcnunes/Computing2018>, 2018. Last accessed 15 July 2018. [cited at page 84, 200]
- [195] Sonarqube.org. <http://www.sonarqube.org>, 2016. Last accessed 3 November 2016. [cited at page 84, 165]
- [196] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14*, pages 42–53, New York, NY, USA, 2014. ACM. [cited at page 86, 183]
- [197] <https://github.com/pjcnunes/EDCC2017>, 2017. Last accessed 7 August 2021. [cited at page 101]
- [198] Sensitivity and specificity. https://en.wikipedia.org/wiki/Sensitivity_and_specificity, 2021. Last accessed 4 June 2021. [cited at page 108]
- [199] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *IN WODA 2003: ICSE WORKSHOP ON DYNAMIC ANALYSIS*, pages 24–27, 2003. [cited at page 135]
- [200] G. Deepa and P. Santhi Thilagam. Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Information and Software Technology*, 74:160–180, 2016. [cited at page 135]
- [201] Dimitris Mitropoulos, Panos Louridas, Michalis Polychronakis, and Angelos Dennis Keromytis. Defending against web application attacks: approaches, challenges and implications. *IEEE Transactions on Dependable and Secure Computing*, 16(2):188–203, 2017. [cited at page 135]

- [202] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. Automated testing for SQL injection vulnerabilities: An input mutation approach. *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, (May):259–269, 2014. [cited at page 135]
- [203] L. Lei, X. Jing, L. Minglei, and Y. Jufeng. A dynamic sql injection vulnerability test case generation model based on the multiple phases detection approach. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 256–261, July 2013. [cited at page 135]
- [204] Rips technologies. <https://www.ripstech.com/features>, 2019. Last accessed 17 April 2019. [cited at page 136]
- [205] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175, Sept 2018. [cited at page 138]
- [206] Chen Ping. A second-order sql injection detection method. In *2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 1792–1796, 2017. [cited at page 144]
- [207] XDebug extension for PHP. <https://xdebug.org>. Last accessed 30 January 2018. [cited at page 144]
- [208] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project. Last accessed 4 February 2018. [cited at page 144]
- [209] Xtm, Xdebug Trace Manipulator. <https://github.com/delins/xtm>, 2018. Last accessed 28 January 2018. [cited at page 145]
- [210] SQLMap. <http://sqlmap.org>, 2018. Last accessed 11 November 2018. [cited at page 148]
- [211] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out doomsday: Towards detecting and preventing dom cross-site scripting. In *2018 Network and Distributed System Security Symposium (NDSS)*, 2018. [cited at page 154]
- [212] Jose D’Abruzzo Pereira, João R. Campos, and Marco Vieira. An exploratory study on machine learning to combine security vulnerability alerts from static analysis tools. In *2019 9th Latin-American Symposium on Dependable Computing (LADC)*, pages 1–10, Nov 2019. [cited at page 163]
- [213] PhpMetrics.org. <http://www.phpmetrics.org>, 2016-10-03. [cited at page 165]
- [214] PHPMD - PHP mess detector. <https://phpmd.org>, 2017. Last accessed 6 January 2017. [cited at page 165]
- [215] Software quality enhancement. <http://www.squale.org>, 2016. Last accessed 3 November 2016. [cited at page 165]

-
- [216] Ömer Faruk Arar and Kürşat Ayan. Deriving thresholds of software metrics to predict faults on open source software: Replicated case studies. *Expert Systems with Applications*, 61:106–121, Nov 2016. [cited at page 165, 166, 172]
- [217] M. Schroeder. A practical guide to object-oriented metrics. *IT Professional*, 1(6):30–36, Nov 1999. [cited at page 165]
- [218] M Sankar and Anthony Irudhyaraj. Software Quality Attributes for Secured Web Applications. *International Journal of Engineering Science Invention*, 3(7):19–27, 2014. [cited at page 165, 166]
- [219] Doaa Nabil, Abeer Mosad, and Hesham A. Hefny. Web-Based Applications quality factors: A survey and a proposed conceptual model. *Egyptian Informatics Journal*, 12(3):211–217, 2011. [cited at page 165]
- [220] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976. [cited at page 166, 171, 173]
- [221] Vu Nguyen, Sophia Deeds-rubin, Thomas Tan, and Barry Boehm. A sloc counting standard. In *COCOMO II Forum 2007*, 2007. [cited at page 166]
- [222] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. *IEEE International Conference on Software Maintenance, ICSM*, 2010. [cited at page 166, 168, 170, 173, 174]
- [223] Tiago L. Alves, José Pedro Correia, and Joost Visser. Benchmark-Based Aggregation of Metrics to Ratings. In *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, pages 20–29. IEEE, nov 2011. [cited at page 166, 170, 172, 176]
- [224] Arthur H. Watson, Thomas J. McCabe, and Dolores R. Wallace. Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric. In *U.S. Department of Commerce/National Institute of Standards and Technology*, 1996. [cited at page 166, 173]
- [225] Paloma Oliveira, Fernando P. Lima, Marco Tulio Valente, and Alexander Serebrenik. RTTool: A Tool for Extracting Relative Thresholds for Source Code Metrics. In *2014 IEEE International Conference on Software Maintenance and Evolution*, number c, pages 629–632. IEEE, Sep 2014. [cited at page 166, 173]
- [226] José Pedro Correia and Joost Visser. Certification of technical quality of software products. In *Proc. of the Int'l Workshop on Foundations and Techniques for Open Source Software Certification*, pages 35–51, 2008. [cited at page 166, 173, 176]
- [227] ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model. http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749, 2001. Last accessed 31 January 2016. [cited at page 167]
-

- [228] Jean-Louis Letouzey. The SQALE Method - Definition Document, Version: 1.0. <http://www.sqale.org/wp-content/uploads/2010/08/SQALE-Method-EN-V1-0.pdf>, 2012. Last accessed 20 January 2016. [cited at page 167]
- [229] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. [cited at page 167]
- [230] Jim A McCall, Paul K Richards, and Gene F Walters. Factors in software quality. volume i. concepts and definitions of software quality. Technical report, DTIC Document, 1977. [cited at page 167]
- [231] Iso 25000 portal. <http://iso25000.com/index.php/en>, 2016. Last accessed 15 December 2016. [cited at page 167, 168, 169]
- [232] Martin Fowler. Is High Quality Software Worth the Cost? <https://martinfowler.com>, 2019. Last accessed 29 May 2019. [cited at page 168, 169]
- [233] Péter Hegedus, Tibor Bakota, Gergely Ladányi, Csaba Faragó, and Rudolf Ferenc. A drill-down approach for measuring maintainability at source code element level. *Electronic Communications of the EASST*, 60, January 2013. [cited at page 169]
- [234] Dennis Bijlsma, Miguel Alexandre Ferreira, Bart Luijten, and Joost Visser. Faster issue resolution with higher technical quality of software. *Software quality journal*, 20(2):265–285, 2012. [cited at page 172]
- [235] Bertrand Stivalet and Elizabeth Fong. Large Scale Generation of Complex and Faulty PHP Test Cases. *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 409–415, 2016. [cited at page 195]

