João Luís A. M. M. Almeida

Using Evolutionary Algorithms to Automate the Correction of Software Vulnerabilities

1 2 9 0 UNIVERSIDADE Ð COIMBRA

1 2 9 0

UNIVERSIDADE Ð COIMBRA

João Luís Araújo Madeira de Meneses de Almeida

Using Evolutionary Algorithms to Automate the Correction of Software Vulnerabilities

Dissertation in the context of the Master in Informatics Security,
advised by Professor Nuno Antunes and Professor Nuno Lourenço and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2021

Faculty of Sciences and Technology

Department of Informatics Engineering

Using Evolutionary Algorithms to Automate the Correction of Software Vulnerabilities

João Luís Araújo Madeira de Meneses de Almeida

Dissertation in the context of the Master in Informatics Security,
advised by Professor Nuno Antunes and Professor Nuno Lourenço and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2021

1 2 9 0

UNIVERSIDADE Ð
COIMBRA

This page is intentionally left blank.

This work was carried out in the Software and Systems Engineering (SSE) Group of the Centre for Informatics and Systems of the University of Coimbra (CISUC).

This work was supported by the project *TalkConnect: Voice Architecture over Distributed Network* (POCI-01-0247-FEDER-039676), co-financed by the ERDF, through Portugal 2020 (PT2020), and by COMPETE2020 and FCT.

This work is partially funded by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the project CISUC - UID/CEC/00326/2020 and by European Social Fund, through the Regional Operational Program Centro 2020.

This work has been supervised by Professor Nuno Manuel dos Santos Antunes and Professor Nuno António Marques Lourenço, Assistant Professors at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

This page is intentionally left blank.

Acknowledgements

This work could have certainly been so much more,
and what it did become was due to the unwavering guidance and support
of my advisors, Professor Nuno Antunes and Professor Nuno Lourenço.
Thank you the privilege of working with/learning from you.

To my friends and family,
thanks for being there for me, even if I was way too distant.
Hoped to make you proud.

This page is intentionally left blank.

Abstract

The overwhelming cost of software maintenance has rallied up the field of automated program repair, looking to free developers from the burden imposed by the continuous discovery of faults. Vulnerabilities are a particularly attractive target, given the potential impact of their exploitation while mostly following common patterns for detection and correction. There is, however, a clear lack of repair tools focusing on vulnerabilities, despite not needing an oracle for detection and having lower patch complexity.

This work proposes an evolutionary framework based on Genetic Programming for the automated correction of vulnerabilities leveraging the corresponding fix patterns, allowing precise modifications in the original source code through its tree-based representation. A population of candidate fixes is evolved, guided by an assessment of their quality that checks whether the vulnerability has been fixed and functional correctness preserved. To deal with the enormous search space of possible source code modifications, we apply domain specific constraints to minimize the generation of invalid (uncompilable) code including the preservation of typing and syntactic correctness. Further, we restrict the evolutionary procedure to specific lines of code, extracted from reports of instrumentation-based tools. The repair process can then become autonomous through integration with existing vulnerability detection tools based on automatic test generation.

This required focusing on a single language, and, initially, on a limited set of vulnerability types. C was chosen due its prevalent adoption for critical software, and propensity for vulnerabilities related to memory safety, ranked as some of the most dangerous. We show our (GPVE) engine's capabilities on a set of vulnerabilities injected into two data structure implementations, while looking to study the impact of their type, localization, use of fix patterns, and of different fitness functions. The engine consistently generates complex fixes, with a 87.9% success rate across 9 vulnerabilities, though often to the detriment of other non-functional properties, namely performance and understandability. Further, the use of reduced test suites allowed for the acceptance of incorrect fixes that had overfitted to its test cases.

Future work will look to apply the engine to large-scale programs where high coverage test suites provide stronger guarantees over the generated fixes' correctness. Despite its limitations, our work nevertheless shows Genetic Programming's applicability when tailored to vulnerability repair, being able to efficiently evolve programs to pass test cases that otherwise revealed vulnerabilities.

Keywords

Automated Program Repair, Evolutionary Computation, Genetic Programming, Software Security, Vulnerability Correction

This page is intentionally left blank.

Resumo

O custo esmagador da manutenção de *software* despoletou o campo de reparação automática de programas, procurando libertar os programadores da carga imposta pela descoberta contínua de falhas. Vulnerabilidades são um alvo atraente, dado o impacto da sua *exploração*, mas seguindo padrões comuns de deteção e correção. Há no entanto uma clara falta de ferramentas de reparo focadas em vulnerabilidades, apesar de não necessitarem de *oráculo* de testes para a sua deteção e terem correções menos complexas.

Este trabalho propõe uma *framework* evolucionária baseada em Programação Genética para a correção automática de vulnerabilidades, aproveitando os padrões de correção correspondentes, e permitindo modificações precisas no código original através da sua representação baseada em árvores. Uma população de correções candidatas é evoluida, guiada por uma avaliação da sua aptidão que verifica se a vulnerabilidade foi corrigida e a correção funcional preservada. Para lidar com o enorme espaço de procura de modificações possíveis ao código, aplicamos restrições específicas ao domínio de modo a minimizar a geração de código inválido (que não compila), incluindo a preservação da correção sintática e de tipagem. Restringimos também o processo evolucionário a linhas específicas do código, extraídas de relatórios de ferramentas baseadas em instrumentação. O processo de reparo pode então ficar autonómo, através da integração com ferramentas existentes de deteção de vulnerabilidades baseadas em geração automática de testes.

Focámo-nos numa única linguagem, e, inicialmente, num conjunto limitado de tipos de vulnerabilidade. C foi escolhida devido à sua adoção para *software* crítico e propensão para vulnerabilidades relacionadas com acesso à memória, classificadas como das mais perigosas. Mostramos as capacidades do *motor* de correções "GPVE" num conjunto de vulnerabilidades injetadas em duas implementações de estruturas de dados, procurando estudar o impacto do seu tipo, localização, uso de padrões de correção e de funções de avaliação diferentes. O motor gera consistentemente correções complexas, com uma taxa de sucesso de 87.9% em 9 vulnerabilidades, embora por vezes em detrimento de outras propriedades não funcionais, nomeadamente desempenho e legibilidade. O uso de conjuntos reduzidos de testes possibilitou ainda a aceitação incorreta de correções sobreajustadas aos seus casos de teste.

Trabalho futuro passará por aplicar o motor a programas de grande escala, com conjuntos de teste de alta cobertura que forneçam garantias fortes sobre a correção das soluções geradas. Apesar duas suas limitações, o nosso trabalho demonstra a aplicabilidade de Programação Genética ao reparo de vulnerabilidades, sendo capaz de evoluir programas eficientemente para passar em casos de teste que de outro modo revelavam vulnerabilidades.

Palavras-Chave

Reparação Automática de Programas, Computação Evolucionária, Programação Genética, Segurança de Software, Correção de Vulnerabilidades

This page is intentionally left blank.

Contents

Acronyms

ASan	AddressSanitizer. xvii, 9, 13, 21, 27, 28, 35–38, 49, 61, 73
AST	Abstract Syntax Tree. 2, 19, 20
CFG	Context-Free Grammar. 14
CI	Continuous Integration. 18
CIL	C Intermediate Language. 13, 25–27, 33, 37, 46
CVE	Common Vulnerability Enumeration. 6
CWE	Common Weakness Enumeration. xvii, 6, 36, 61, 73
DSGE	Dynamic Structured Grammatical Evolution. xv, 16, 17
EA	Evolutionary Algorithm. 1, 10–16, 18, 19, 21, 22, 31, 51
GE	Grammatical Evolution. 15, 16, 22
GGGP	Grammar-Guided Genetic Programming. 13, 14, 16, 26
GI	Genetic Improvement. 1, 17–19
GP	Genetic Programming. xv, 2, 12–14, 16, 17, 19, 21, 25, 41
GPVE	Genetic Programming for Vulnerability repair Engine. xvii, 2, 25, 27, 28, 30–33, 35–37, 39, 41–43, 45–47, 51
SBSE	Search-Based Software Engineering. 17
SMT	Satisfiability Modulo Theories. 7, 19
STGP	Strongly-Typed Genetic Programming. 13

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

Software maintenance constitutes a major portion of a project's budget [1], with inadequate testing infrastructure being estimated in 2002 to cost $59 billion annually [2]. Security vulnerabilities are particularly worrying given the impact of their exploitation and the fact that most developers are not specialized in security, thus increasing the likelihood of introducing and the cost of fixing them. This has led to companies adopting bounty programs, outsourcing vulnerability detection and correction, from an inability to deal with their inevitable presence [3, 4, 5].

However, vulnerabilities and their fixes tend to follow common patterns that have been extensively documented [6, 7], making the automation of their correction, guided by this collected knowledge, an inviting proposition. In fact, automated program repair is an established field, with the majority of implementations being based on Genetic Improvement (GI) [8] - the application of Evolutionary Algorithms (EAs) to repair or improve the performance existing software.

Nevertheless, scalability, usability, and applicability issues [9] leave plenty of room for improvement, with no current tool seeing mainstream use. One likely cause is the desire to fix all software faults, whereas focusing on software vulnerabilities allows one make several key assumptions about the repair process. Regarding the generation of fixes, vulnerabilities can be detected at runtime through instrumentation, being reported at a precise location rather than having to consider the entire program. Further, vulnerability patches require fewer changes than bugs, and are restricted to a single function 59% of the times [10]. Regarding the correctness of generated fixes, the original program may be used to generate a high-coverage test suite with expected outputs, seeing as its functionality should be preserved [11].

The appearance of such a vulnerability repair tool would free up developers to focus on more immediately pressing concerns - delivering the desired functionality. Furthermore, it would ensure some level of end-of-life support for the program, when the team migrates to another project. With this work, we hope to contribute to eventually achieving *"true industrial application of program repair"* [12].

1.1 Approach

With this in mind, our mail goal is to build a scalable, Genetic Programming (GP)-based, automatic vulnerability repair tool leveraging known patterns. We are operating directly on the Abstract Syntax Tree (AST)'s individual nodes, leading to an enormous search space. To deal with this, we explore the following approaches:

- Focus on extracted snippets of code where the vulnerability is likely to be, through localization;
- Preserve syntactic/semantic correctness by generating candidate fixes according the language's grammar, and restricting its productions according to typing rules, and variables in the current environment;
- Definition of vulnerability specific grammars, by adding parametrizable fix templates to its productions. These templates should be easy to add, so that vulnerability support can be extended;
- Usage of vulnerability-specific variation operators, offering an alternative way to support fix patterns. For example, missing `if` constructs, a common mistake that causes vulnerabilities [7], can be efficiently corrected with the wrapping mutation described in [13];
- Define dynamic fitness functions, possibly through coevolution of the tests with the candidate fixes. Such a function would then be tuned to efficiently discriminate fixes, leaving more time for the exploration of the search space.

Some trade-offs will be taken into consideration. We will focus on C programs, to allow us to encode domain-specific knowledge that is essential for the problem's tractability. The reliance on fix patterns means only a limited set of memory-access vulnerabilities types will be initially supported. We are relying on a report of where the vulnerability is exercised, but if the root cause was, for instance, an incorrect initialization, then we may end up correcting all usages which is not the expected fix. And other implementations based on the assumption that the fix exists elsewhere in the source code, and as such rely on a *cut-paste* approach, may have an advantage in efficiency. Our approach, on the other hand, benefits from being able to represent a fix if it does exist.

1.2 Contributions

This work's novelty lies in tailoring GP to vulnerability repair, as opposed to any bug. Its main contributions are the:

- Study into the suitability of GP for vulnerability correction, culminating in the proposal of a repair framework leveraging existing vulnerability detection tools.
- Implementation of a GP based correction *engine* named Genetic Programming for Vulnerability repair Engine (GPVE) for memory-access vulnerabilities in C programs, looking to apply their common fix patterns while minimizing the generation of invalid code.
- Experimentation on a set of injected vulnerabilities, including analysis of the expected *cost* of fixing each, highlighting the engine's efficiency despite the search space size.

Auditing the proposed fixes revealed how preference for smaller fixes adversely affected the program's performance and understandability, as well as fixes overfitting on the reduced test suites.

1.3 Structure

The remainder of this document is structured as follows.

- Chapter 2 presents the background concepts needed to understand this work.
- Chapter 3 lays out the proposed framework, giving a high-level overview of each component and the approach taken towards the main challenges it faces.
- Chapter 4 details the implementation of the vulnerability correcting *engine*, also looking at its usage and performance.
- Chapter 5 describes the vulnerabilities injected into two data structure implementations, on which several experiments were run to evaluate the engine's ability to fix, and the impact their type, location and associated fix patterns had.
- Chapter 6 reports on the main results and analyses some of the proposed fixes.
- Chapter 7 sums up the key takeaways and outlines directions for future work.

This page is intentionally left blank.

Chapter 2

Background

2.1 Software Security

Software Security [14] is related to its ability to function properly under malicious attack [15]. We are concerned with the preservation of the so-called *CIA triad*, consisting of the following three properties [16]:

- Confidentiality: prevent unauthorized disclosure of information
- Integrity: prevent unauthorized system modification
- Availability: prevent malicious resource exhaustion (*"readiness for correct service"* [16])

Security incidents can have devastating consequences as software is used to control critical infrastructures. Further, these systems are ever more exposed [14], while being extremely complex, so that a developer can introduce defects from inability to reason over its entirety. For attacks to be successful there must exist the corresponding vulnerability looking to be exploited in the system [14]. But an attacker as a plethora of easy-to-run exploits available (eg. Metasploit[1]), and the developer needs to be aware of each or risk inadvertently opening up an attack vector for it.

Security should be a main concern even after deployment. A system is never *perfectly secure* - new exploits are constantly discovered targeting what would have previously not even been considered a vulnerability. Still, automatic repair helps lessen the developer's burden, requiring minimum input for fix acceptance and/or feedback.

2.1.1 Vulnerabilities

Vulnerabilities are exploitable faults, which allow attackers to harm the system [16] by violating one or more security properties. A fault is the cause of deviation in the system state (error) that could lead to incorrect system service (failure) [16]. In general, there are several ways to deal with faults [16]: prevention (eg. follow best practices), tolerance (ensure correct service in the presence of exercised faults), or removing faults through repair.

Vulnerabilities are language dependent, with low-level languages like C suffering from memory safety from permitting direct memory access - this is a poisoned gift: on one hand giving

[1] `https://www.metasploit.com/`

us great flexibility and power, while on the other allowing a myriad of related vulnerabilities. In the Chromium project, around 70% of high-severity vulnerabilities were found to be related to memory safety [17]. And while recent languages, namely Rust, attempt to solve these issues, there is a tremendous amount of legacy code written in C for security-critical software - operating systems, databases, browsers, interpreters for higher-level languages - on which new vulnerabilities keep being discovered, and where the effort of re-implementing in a safer language would be insurmountable.

A particularly interesting example of a memory safety-related vulnerability, since it affects all three CIA properties, are buffer overflows due to C's lack of bound checking on memory access: these allow one to read/modify unauthorized memory locations, which could lead to denial of service if the operating system detects it (segmentation fault). This highlights the possible trade-off between security and performance. Not bound checking could be reported by automatic tools as a *code smell* (indication of defect), potentially exploitable, however we are only sure of that if it is verified on an actual input. Should languages then *trust* the programmer that it never occurs (indexes are known to always be valid)? Or rather check no matter no matter how straightforward the array access is (incurring in slight performance hit)? As opposed to C, OCaml does this by default - but can be turned off with the `-unsafe` compilation flag.

2.1.2 Vulnerability Patterns

The FindBugs tool [18] checks code for a set of bug patterns - code idioms likely to be bugs - encoded through analysis of previously identified bugs. Simple pattern detectors were implemented and found to generalize in finding bugs in real applications, highlighting the practicality of these approaches.

An analysis of software bugs encoded 27 bug+fix patterns and found that these covered between 45.7% and 63.6% of those existing in seven open source projects [5]. Furthermore, the frequency of each pattern was found to be similar across projects. This shows that a limited number of patterns can nevertheless generalize to the majority of bugs.

Similarly to faults, the common characteristics shared by a set vulnerabilities can be encoded into vulnerability patterns [6]. We can extract these from publicly available datasets collecting vulnerabilities like Common Vulnerability Enumeration (CVE) [2], which can be classified according to the corresponding weakness type, enumerated in the Common Weakness Enumeration (CWE) [3]. These CWEs are typically accompanied by examples and mitigations, thus serving as a valuable tool in extracting patterns for their detection and correction.

A 2020 ranking of weaknesses according to a combination of the frequency and severity of recent CVEs is given in [19] (updated annually). Here, memory safety issues are among the top CWEs - out-of-bounds read/write at #4 and #2 respectively, use-after-free at #8,... This list therefore help us prioritize which to support for automated repair.

While the purpose of CWEs is primarily educational, SARD[20][4] provides datasets of security errors and corresponding fixes aimed at developing software assurance tools. Testing such tools is thus facilitated by samples from both synthetic (train) and production (test) code. A dataset [21] of 291 buffer overflows, where each has a correct and three vulnerable versions (differing by the egregiousness of the overflow) is particularly enticing. These

[2]`https://cve.mitre.org`

[3]`https://cwe.mitre.org`

[4]`https://samate.nist.gov/SARD`

are simple snippets, varying according to numerous parameters like the index complexity or memory location, that if incorporated in larger programs with significant functionality would allow us to test the proposed end-to-end framework.

[7] gathers the most frequent mistakes leading to software vulnerabilities. These include missing `if` constructs or function calls, and help us identify the fix patterns that should be supported and efficiently generated by the repair engine (eg. wrapping statement(s) in `if` construct). Moreover, the authors identify that 43.5% of the studied vulnerabilities result from a single mistake, and around 75% from three or less [7]. This knowledge helps determine the expected behaviour from the search-based repair procedure, perhaps bounding the number of modifications a candidate fix is allowed to undergo.

The number of vulnerabilities supported for automated repair is typically limited at first, targeting particular prevalent classes [22]. However new vulnerabilities are constantly discovered [6], that may not fit existing fix patterns, therefore it is important that the framework can be easily extended with new patterns.

2.1.3 Vulnerability Detection

In this section we describe several automatic test generation techniques which double up as powerful vulnerability detection when paired with code instrumentation. Test cases are essential for assessing a candidate fix's quality in automated program repair.

We want to generate high path-coverage test suites for an arbitrary program, and as such the focus will be on automated techniques that can be easily integrated in the project's framework.

As Dijkstra famously put it: *"Program testing can be used to show the presence of bugs, but never to show their absence"* [23]. Given the potential size of the input space, a subset must be considered. Still, test suite's completeness is not even desirable (as running it would be similarly infeasible), just its ability to accurately detect deviations from correct functionality.

Symbolic Execution

Symbolic Execution [24, 25] differs from traditional testing by not supplying the program with concrete inputs. Rather, it considers symbols representing arbitrary values that will be constrained according to the current path being exercised. That is, execution is forked upon reaching a branch, considering the corresponding constraints in turn, with the path only being followed if the constraints were found to be solvable - typically done through Satisfiability Modulo Theories (SMT) (eg. z3 [26]). If a path terminates, concrete inputs are then generated by again solving the constraints - this ensures reproducibility and the absence of false positives, as opposed to static analysis, cf. Sec.2.1.3 [27].

The big advantage to this approach is that, while the input domain may be infeasible to test, the number of distinct paths in a program is often much more tractable. We can then exhaustively look at each such path, while simultaneously considering the (possibly infinite) set of inputs that satisfy it.

This search leaves room for heuristics when choosing which path to explore next. EXE's [27] in particular (a precursor to KLEE [28]) opts for those leading to underexplored branches/staments so as to increase code coverage.

As such, symbolic execution appears to be a great approach for generating high coverage test suites, necessary for automated repair if we are to ensure the fixes' correctness.

However, this is no silver bullet however since path explosion is a real issue in complex programs, coupled with the potentially slow/incapable constrain solving. Furthermore, the use of libraries and interaction with the environment can limit its applicability by not possessing all the code. Scalability is a major concern [25].

Regarding vulnerability detection in particular, the idea would be to look for crashes, likely indicators of memory safety violations in C programs.

KLEE [28] is a symbolic execution engine operating LLVM code, to which C code can be compiled. The program must be pre-processed, by specifying which variables should be made symbolic. From there, the search can be guided by providing assumptions about the variables values to constrain the search space, or preferences to bias the search engine. In an 89 hour run, KLEE was found to beat the heavily tested `GNU coreutils`'s suite.

Furthermore, in the context of program repair, functional equivalence can be verified between the original program (`f`) and a possible fix (`f''`) can be guaranteed through the assertion `f(x)==f'(x)` [28] - KLEE will then take care of checking this over all possible inputs `x`. However, this should be reserved for a final acceptance step, given the prohibitive cost of symbolic execution.

Fuzzing

Fuzzing [29, 30] continuously tests a program with modified inputs (randomly or by heuristic-based mutations) to find security vulnerabilities.

Coverage-guided fuzzers, with AFL [5] being the seminal implementation, are of particular interest, whose corpus of generated tests avoid path redundancy, while the search aims for unexplored parts of the code - this is achieved by instrumenting the code to track coverage.

Driller [31] augments fuzzing with symbolic execution when the fuzzer is found to be stuck in a certain path, allowing it to continue by solving the associated constraints. An example, described in [27], would be an equality conditional over 32-bit integers - easy to solve through SMT but highly unlikely that a random fuzzer will stumble upon the precise value. On the other hand, it largely avoids symbolic execution's path explosion problem.

NAUTILUS [32] considers structured input according to a grammar, where otherwise many tests would consist of syntactically invalid inputs - immediately rejected and thus hindering path coverage.

Two open-source tools standout with great results (*trophy cases*): AFL and libFuzzer [6] which can be used through the `clang` compiler's `-fsanitize=fuzzer` flag. A key difference between the two is that while AFL runs the mutated inputs on a newly-forked program instance, libFuzzer does it in-process. This gives a significant speedup but also means that fuzzing stops on inputs that crash - undesirable when fishing for vulnerabilities, since we simultaneously look to attain a high-coverage test suite used for validating repairs. The solution suggested by the developers, to fix the bug so that fuzzing can restart hinders its use for automated correction.

Regarding structured input, AFL has limited support - only dictionaries of tokens as far

[5] `https://lcamtuf.coredump.cx/afl/`

[6] `https://llvm.org/docs/LibFuzzer.html`

as the author is aware. LibFuzzer, meanwhile, supports protobuf specifications (similar to traditional grammars) with a tailored mutation library [7]. AFL, however, is commonly extended by researchers, and Superion[33] added support for ANTLR-specified grammars.

The plug-and-play nature nature of coverage-guided fuzzers make them great options for integration into an end-to-end automated repair framework. These approaches are, however, held back by mostly supporting only vulnerabilities that lead to a crashes during execution.

Sanitizers

Instrumentation-based tools for runtime checking are a great way to *awaken* dormant faults, i.e., that would otherwise not cause an error [16]. Returning to the buffer overflow example, if we read a few bytes off the end of the buffer the program will likely continue as *normal*, only failing when it attempts to access OS-protected memory (larger offsets). With these instrumentation tools, buffer boundaries would be marked so that any invalid access would be immediately reported.

Valgrind [34] is one such popular tool, based on shadowing memory values with a description of their status. Recently, however, Google's AddressSanitizer (ASan)[35] has emerged with increased performance (average slowdown of 2x vs. Valgrind's 20x) and ease-of-use - `fsanitize` compilation flag on `gcc`/`clang`. It is similarly based on compiler instrumentation for checking the compacted shadow state, and a specific runtime for allocating/freeing memory, so that invalid accesses are detected - memory on buffer boundaries is *poisoned*, as is that of freed buffers.

With ASan, detected faults immediately crash the execution, with a detailed output containing its classification according to the type of violation observed - including buffer overflows and use-after-free's. Localization can then be inferred through tracing the coverage of the associated negative test, either through the `gcov` utility, or `clang`'s SanitizerCoverage.

There are other similar sanitizers [8] for increasing the scope of supported vulnerabilities - undefined behaviour, concurrency vulnerabilities (data races and deadlocks), etc.

These tools thus form a great pairing with automatic test generators, turning them into powerful vulnerability detectors.

Static Analysis

Static Program Analysis aims to reason about a program's behaviour without executing it [36]. Akin to manual audits, but the programmer instead encodes their knowledge of a vulnerability's pattern into an automated procedure.

Benefits include the time saved, integration in the development process / with code editors, possible prioritization of the different reported faults, and that by reasoning about the source code, the location is always tied to the fault report [37].

However, according to Rice's theorem,all non trivial properties about a program are undecidable (a famous precursor being Turing's proof of the halting problem's undecidability), following that approximations must often be considered [36]. This limits their precision, entailing the likely reporting of false positives, essentially wasting the automated repair's

[7]`https://github.com/google/libprotobuf-mutator`
[8]`https://github.com/google/sanitizers`

time, and false negatives, giving a false sense of security and hindering the (idealistic) goal of perfect security.

Furthermore, static analysis typically generates error messages that require expertise to be analyzed [37], these are great for quick programmer feedback during development [38] but pose a challenge for integrating it with automated repair, while dynamic approaches produce a failing test case from which classification and localization easily ensues.

Looking at what developers find lacking in static analysis tools [37] helps set goals for our automated repair:

- Integration with the workflow, suggesting quick fixes tied to recent code commits, possibly through automatic pull requests.
- Understandability of the results - modifications introduced by fixes should be minimal and obvious, with the overall code remaining familiar.
- Ease of configuration - static analysis tools suffer from not being able to reduce the volume of false positives. For automated repair we may concern ourselves with bombarding the programmer with too many candidate fixes (specify acceptability standards).

2.2 Evolutionary Algorithms

Evolutionary Algorithm (EA) encompass several automatic problem solving techniques inspired by the process of evolution through natural selection [39]. These methods strive to iteratively improve a population of candidate solutions, wherein the selection of *parents* from which new solutions are generated is guided by an assessment of their quality - the fitness function.

These are typically applied to *hard* problems, that is, with no feasible polynomial-time algorithm. Moreover the shape of the optimal solution and the search space is unknown [40] - we can't follow the gradient towards optima (as is the case with backpropagation, popular for ANNs). Our knowledge is essentially limited to generating candidate solutions and evaluating their fitness.

These algorithms are necessarily reliant on some stochastic components - deterministic algorithm would be far too slow (brute force is infeasible while EAs can output good enough solutions in acceptable time (anytime behaviour [39]). Success is then achieved by balancing two important concepts: **exploration** of the full search space which is related to avoiding getting caught in local optima (should not tunnel-vision onto only the fitter individuals) with **exploitation** - guiding a solution by preferring to *follow* fitter solutions

The typical EA is summarized by Alg.1.

Algorithm 1: Generic EA, adapted from [41]

```
Initialize population;
repeat
    Evaluate fitness;
    repeat
        Select parent(s);
        Breed offspring;
    until Next generation is complete;
until Acceptable solution or stopping condition is met;
```

First of all we must consider the **representation** of solutions - how they are manipulated by the EA - genotype versus their acual *shape* - phenotype The choice of genotype representation is varied and commonly consists of lists, trees, ...

Initialization typically consists in generating a certain number of random candidate solutions corresponding to the size of the population - an important parameter in influencing the degree of initial exploration. In some cases, additional knowledge of the problem domain could entice seeding the population with prepared individuals which are believed (or in previous runs of the EA were found) to be relatively fit and help in kickstarting the search procedure.

The fitness **evaluation** aims to assess an individual's quality and compare it with others in the population - so that the primary focus of the search procedure is on the fitter individuals (seeking to improve from one generation to the next). An intermediate step mapping genotype to phenotype may be needed. Here, an important concern is that there may be invalid solutions. Approaches including attributing a penalty or very poor fitness, as for example a negative value if only positive ones were expected.

Selecting parents, the general idea is that fitter individuals should be more likely (ie. higher probability, not guaranteed). A popular technique, that will be adopted in this work, is tournament selection, where the most fit out of N randomly sampled individuals is selected. In theory this means that the majority of individuals (all but the $POP\_SIZE - N$ least fit) can be selected, and that the absolute difference between fitness values is not a concern - or else highly fit individuals could be overwhelmingly selected, undesirably(leads early convergence) reducing population diversity [41].

There are two common variation operations for **breeding** new individuals from existing: **mutation**, which introduces new genetic material in an individual, typically by replacement, and **crossover** which recombines genetic material from two individuals. The inner workings of each are obviously dependent on the representation that was chosen. And again these operations are stochastic - it is expected that the same individual could be selected multiple times for reproduction. An associated concern is when to apply each - a probability is typically associated with each, potentially mutually exclusive (one but not both).

The next generation needs not be composed solely of newly bred individuals. It is also common to have **elitism**, wherein the N most fit individuals are preserved in the next generation, preventing them from being discarded. This adds another degree of exploitation that could lead to premature convergence - care needs to be taken to ensure diversity [40].

Finally, this process **terminates** either when a viable solution is accepted, or as is often the case, when the best found thus far is returned due to lack of iterations or convergence of population from lack of diversity

In an EA, the representation of solutions shapes the search space, and how we move through it is determined by which individuals we choose to breed - guided by the fitness function's evaluation that identifies good points/regions - and how the new individuals are generated by applying the variation operators. Since this movement relies on heuristic-based mechanisms that resort to some degree of randomness, results aren't guaranteed. In fact, it is possible that due to a lack of understanding of the problem domain, it is not possible to represent the ideal solution, or with an inadequate fitness function the search process could end up no better than random (extreme example: *needle* in a binary function *haystack*).

The parametrization of these mechanisms is important. We need to specify the number of individuals in the population, the size of tournaments in selection, the probability of applying mutation/recombination,etc. Efforts should be taken to ensure that results are reproducible, including reporting the seeds used for random number generation.

The idea of the no free-lunch theorem is that a general algorithm will never be better than a specialized algorithm on particular inputs. However this specialized algorithm, for vulnerability correction, would require a tremendous effort to work on anything other than toy/tailor-made examples. An EA introduces general heuristics to improve upon purely random search, and can furthermore be constrained for the current problem domain through problem specific heuristics [39] - specific representation and structure-aware operators

2.2.1 Genetic Programming

Genetic Programming (GP) [42] is an EA operating on tree-like structures, and thus commonly adopted for the automatic evolution of computer programs The reason being that programming languages are specified by grammars, and code is easier to manipulate if parsed to its derivation tree (structured) rather than as a string (cf. Fig.2.1).

This is a domain-independent method since we only need to change the symbols that a tree node can take. In the original implementation, these symbols are typically divided into functions (internal nodes) and terminals (leaves). There is no need to specify the structure that these symbols must follow, and hence can be applied to domains that are not well understood.

Initialization expands function symbols until a terminal is reached Variation operators follow by selecting a subtree in the parent(s). Mutation generates a new one in its place. Crossover replaces it with the other parent's.

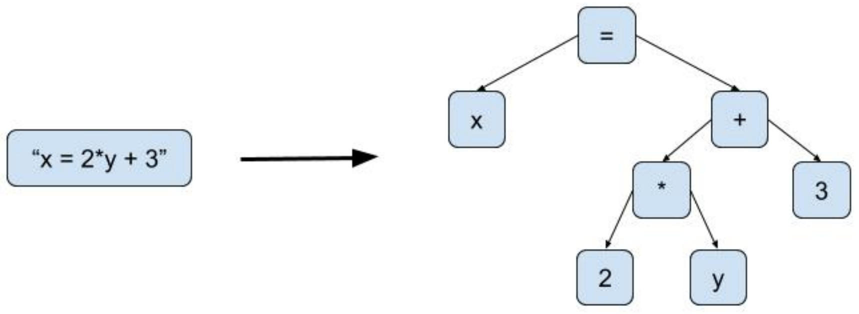


Figure 2.1: Tree representation makes code amenable to structural manipulation

GP has had success in a wide range of applications with human-competitive results (cf. Humies) or better [41]. However the dream of automatic programming remains an open problem with no feasible approach [11] - it can be applied to small domain-specific languages for problems such as artificial ant, but general purpose programming is tremendously challenging due to the size of the problem space.

Nevertheless, we believe automatic vulnerability correction demonstrates many of the properties associated with success in GP [41]:

- test data can be automatically generated with great efficiency, reused throughout runs;

- testing amounts to compiling the code and running on these;
- static methods are inherently limited (Rice theorem,etc);
- an approximate solution that fixes the vulnerability while inadvertnly messing with correctness is valuable, since easier for a developer to fix (ie. good enough solutions can be accepted);

Since the representation is variable-length, a significant concern is the uncontrollable growth of individuals when unaccompanied by corresponding fitness improvement - **bloat** [43]. Particularly in programs, there tend to be several ways to achieve the same goal, and so redundant constructs can appear. The typical approach when generating individuals, or operating on them, is to limit the depth of the resulting trees (once reached there can only be terminals) - which makes the choice of this parameter critical.

There are three additional properties that are important to consider, with the latter typically grouped **closure** [41]:

- **sufficiency** - can we represent a solution with available symbols? When dealing with programs in our project we use C's grammar to guarantee this, albeit with minor modifications to remove syntactic sugar (alternative representations for the same construct, eg. looping) that contribute to redundancy. These modifications mostly come from C Intermediate Language (CIL)[44], a C parser tailored for program analysis and source-to-source transformations.
- **type consistency** - to avoid programs that don't compile. In traditional GP applications, nodes tend to be associated with a single unique type (commonly floats) for removing this concern, or have typing constraints explicitly embedded in the grammar's productions. For general purpose programming though, such solutions are infeasible - will need to define typing rules that restrict the current nonterminal's expansions.
- **evaluation safety** - avoid undefined behaviour, eg. divide-by-zero which traditional GP solves by defining protected functions. In our case, undefined behaviour should be treated as a vulnerability, using a sanitizer similar to ASan - UndefinedBehaviorSanitizer [9]. The reason being that such behaviour is not specified by the language standard and so could allow for exploitation.

Grammar-Guided Genetic Programming

Grammar-Guided Genetic Programming (GGGP) [45] [46] incorporates grammars into the EA, so that individuals amount to derivation trees respecting the grammar's syntactic restrictions, which are preserved throughout the variation operations. For crossover this is achieved selecting subtrees with the same root symbol in both trees.

Moreover we can enforce **type correctness**, commonly called Strongly-Typed Genetic Programming (STGP) [47]. For languages such as C, typing cannot be enforced solely with a grammar, example: in an invocation, the arguments' types are dependent on the function that has been selected, as is also the case with operators. Will thus need to restrict the expansions applicable to a given nonterminal in the grammar, according to its type and associated typing rules. All of this eliminates clearly invalid solutions that would not even compile, making the problem more tractable [48].

[9]`https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

An important concern, so as to minimize performance penalty in the EA loop, is that we already possess all the information needed to correctly expand a node when we so wish to do it. Although expansion follows depth-first order, this must not the case when we reach a binary operation of the kind `exp ::= exp op exp`, where `op` in fact acts as function taking the surrounding `exp`'s as arguments. Functions must be expanded first so that their arguments can be properly constrained. When these arguments are polymorphic, eg. for equality, each argument also constrains the next.

Regarding the choice of mutation/crossover point, being a tree it'll naturally tend to select nodes further down (more common) resulting in typically "smaller" changes, however nothing impedes the selection of a node close to the root - this is an example of balancing exploration vs. exploitation described in [40].

We are thus constraining the search space, which could raise the problem of excluding valid solutions - we are however following C's syntax and semantics so no issue should arise..

We commonly use Context-Free Grammars (CFGs) which can be encoded by the tuple $\{N, T, S, P\}$:

- N being the set of nonterminals (internal nodes),
- T the set of terminals (leaves),
- S the start symbol (root).
- P the set of productions [10] mapping elements of N to lists of elements in $(N \cup T)$ - ie. "we rewrite/expand Ns until a T is reached", note that the same terminal can have several expansions, possibly recurrent being able to generate "infinite" programs (give example? of list -> el list | _)

and which we'll represent through Backus-Naur Form, as exemplified below for the grammar used in [49] for the symbolic regression of the quartic polynomial $(x^4 + x^3 + x^2 + x + 1)$

Note that non-terminals are enclosed in <...> meanwhile terminals are single-quoted. Also, in our implementations thus far this notation is slightly altered so as to integrate typing information and syntactic sugar.

$\langle start \rangle$ ::= $\langle expr \rangle$

$\langle expr \rangle$ ::= $\langle expr \rangle$, op, $\langle expr \rangle$
  | ‘`(`’, $\langle expr \rangle$, $\langle op \rangle$, $\langle expr \rangle$, ‘`)`’
  | $\langle pre\_op \rangle$, ‘`(`’, $\langle expr \rangle$, ‘`)`’
  | $\langle var \rangle$

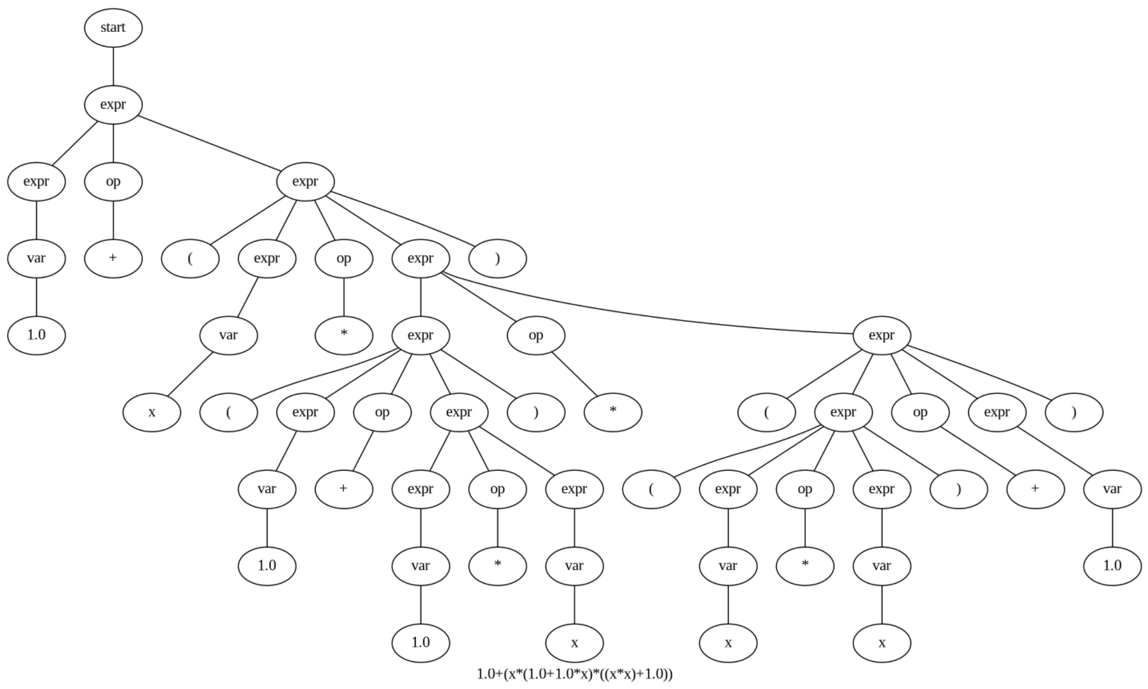
$\langle op \rangle$ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’

$\langle pre\_op \rangle$ ::= ‘`sin`’ | ‘`cos`’ | ‘`exp`’ | ‘`inv`’ | ‘`log`’

$\langle var \rangle$ ::= ‘`x`’ | ‘`1.0`’

A solution to this problem is shown belown, solved with GGGP. Important to note that this is just one of infinite (if no depth limiting) solutions - can apply the commons laws of commutativity, distributivity, associativity,... to have equivalent correct solutions, and also have redundant constructs like a $< exp > +(1.0 - 1.0)$

[10]In this work “expansion” is often used when referring to the productions associated with a specific nonterminal, since this and “terminal” were the constructor names used for the GP engine's inductive tree type (basis of the genotype's representation)



2.2.2 Grammatical Evolution

Grammatical Evolution (GE) [50] is another approach for the automatic evolution of programs, distinct by its use of binary strings for the representation of individuals (**genotype**). These are mapped into the grammar's production rules that generate the derivation tree (**phenotype**), used for fitness evaluation.

A great advantage then is that changing problem domain can be as simple as changing the grammar [49] (only affecting the output of the mapping from rules to tree), while the search procedure can stay the same, which does not even necessarily need to be based off of EAs (other stochastic optimization approaches could serve). Moreover, the linear representation allows the use of variation operators studied in-depth from their application to important problems like the knapsack and travelling salesman (these are NP-complete problems meaning NP problems - the common targets for EAs - can be reduced to them in P time).

But this traditional approach has some issues [51] related with **locality**, the desire for small modifications in the genotype reflect proportionally on the phenotype , and **redundancy**, different genotypes mapping to the same phenotype. An algorithm with low locality will struggle to guide the population throughout the search space, and especially around optima [51], as is expected from EAs. Meanwhile high redundancy can lead to extraneous individuals being generated when an identical in practice has already been evaluated.

For GE an individual may be incomplete - where during mapping its genotype is fully parsed while unexpanded nonterminals remain in the phenotype. The solution chosen by the authors [50] is to wrap around, that is continue parsing from the genotype's beginning. This leads to distinct parts of the phenotype being potentially affected by the same genes, with no further consideration over when this could make sense. A small change in the genotype could therefore greatly affect the phenotype, violating the desired locality principle.

Furthermore the mapping process leads to redundancy - since each codon is 8 bits from the binary string representation, they must compressed into the number of productions available for the current nonterminal through the modulo operator (there are thus numerous ways to represent the same exact production rule).

Locality issues also arise from variation operators, which if not careful will be just as likely to manipulate the beginning of the list which drastically changes the resulting phenotype - all the remaining genes will (likely) be operating on different non-terminals due to changing the initial expansions.

GE has had much success in various problem domains, but not in code generation for general-purpose languages - locality and redundancy issues limit applicability. [52] claims to evolve programs in C but the actual grammar is extremely restricted and only supports the domain of the Santa Fe Ant Trail problem. We strive to support the full C grammar (bar some syntactic sugar) so that whatever the vulnerability in question may be, our framework can in theory generate the solution which corrects it.

Dynamic Structured Grammatical Evolution

Dynamic Structured Grammatical Evolution (DSGE) [49] looks to fix these issues in GE by keeping a list of genes (production rules) for each nonterminal -now, each codon directly corresponds to a production of the associated terminal, removing the need for the modulo operator and therefore reducing redundancy. Moreover, it avoids the wrap around problem by continuing to generate new codons for the non-terminals missing them. This however leads to the need for limiting the choice of recursive productions after a certain tree-depth threshold (control bloat). This is a key choice - it impacts the size of the search space, the possibility of bloat, and whether the solution can be derived.

An initial study for this project was conducted using a slightly modified version of DSGE's standard implementation [51][49].[11]. Problems arose when exploring the crossover operator, since its standard implementation consisted of coin flipping over which parents' codons to take for each symbol. In the context of code this seemed to deteriorate locality, and make it difficult to trace the evolution process as each parent effect would be spread over the entire phenotype (cf. Fig.2.2). The desired behaviour would seem to be "as mutations accumulate, the resulting phenotypes gradually diverge from the deriviation tree of the original solution" [51] Even if very localized modifications can largely impact the functionality of a program, the phenotype not varying much is nevertheless desired for explainability reasons (easier for programmers working with the tool to understand the EA process).

The common GP-like subtree crossover appeared to intuitively make more sense, but traversing DSGE's linear representation as a tree was cumbersome, and counter-productive given the idiomaticness of functional programming languages such as OCaml for tree manipulation (due to their algebraic data types and powerful pattern matching). Such a crossover was also described in [53] as “LHS Replacement Crossover” for the traditional GE representation, concluding its performance gains over the standard, more destructive, implementation.

A GGGP approach was henceforth adopted, for its natural representation when considering code, all the while adopting concepts from DSGE - namely crossover points being selected for the same symbol, and maintaining a mapping process from the AST to compilable (re-add tokens that were abstracted) and readable (indentation) ie. superfluous details when manipulating the program's workings).

[11]available at: https://github.com/nunolourenco/sge3

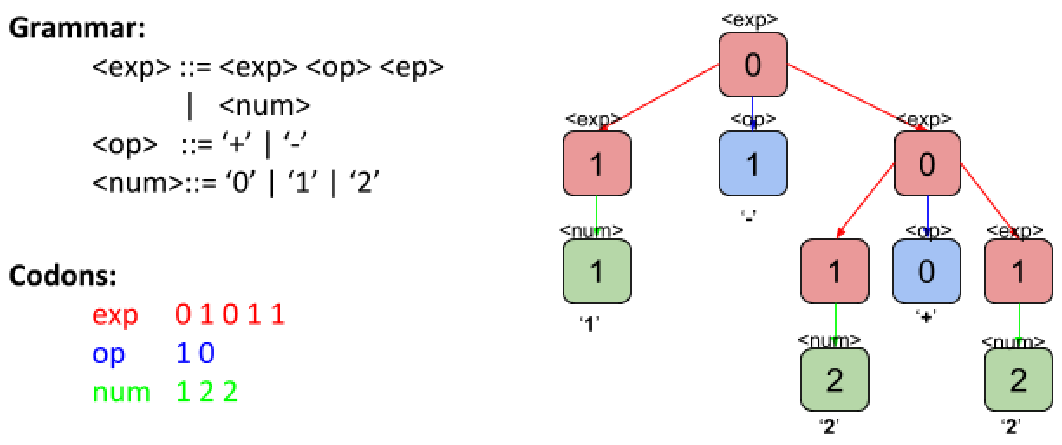


Figure 2.2: DSGE linear representation maps directly to a tree, and we see how each symbol's genes are spread out

DSGE's successes over traditional GE, highlighted in [51], reinforce our positive outlook on a structured/constrained GP.

2.3 Genetic Improvement

Genetic Improvement (GI) [8] is an area of research that builds on existing programs, rather than starting from *scratch* as tends to be the case for automatic programming with GP. It can be seen as a subfield of Search-Based Software Engineering (SBSE) - solving software engineering problems through "search-based metaheuristic optimization techniques" [54].

GI's goal is to make existing programs better with respect to given criteria, typically non-functional properties, which allow fitness evaluation to amount to comparing the modified's outcome with the original's on a series of generated inputs (generate-and-validate patches). It thus side-steps the oracle problem [11] since the functionality should be preserved, the original program is the *de facto* specification [55]. It can also be used for improving functional correctness by fixing bugs if provided with test cases that reveal them - that the original program fails on.

Furthermore, GI can help balance several conflicting non-functional properties by not outputting a single solution but rather the Pareto front, leaving the decision of which to accept to the developer [48].

Given the size of the search space in GI (number of possibly improved programs), the representation is typically a list of patches - commonly inserting/replacing/removing a line of code. This is based on studies that human written code is repetitive and as such the fix exists somewhere else in the codebase [55]. With this representation, it is also straightforward to perform *post-processing cleanup* to the resulting solution so as to remove extraneous edits and facilitate the understanding of the solution - this is done by attempting to remove each edit and re-evaluating the solution [55]. The final fix should be as simple as possible to increase the tools' usefulness in real-life projects, as to reduce time needed for programmer to understand and approve.

Nevertheless, operating on the actual syntax tree offers higher flexibility and granularity, ensuring that the solution can always be represented. There is however a tradeoff between the representation and size/complexity of programs to be evolved. To maximize the applicability to larger programs, fault localization data is needed to extract the excerpt of code likely to cause the vulnerability and significantly reduce the search space. The drawback here is that several runs may be therefore needed with increasing scope of the extracted code until a fix is possible, or a human programmer could intervene with feedback (*human-in-the-loop*).

An obvious problem throughout GI approaches is the demanding fitness evaluation - we can only be sure that correctness was preserved through full path-coverage. But this is simply too computationally demanding for anything more than toy examples. As such, a subset of the test suite should be used and minimized to the extent that it still allows discriminating candidate's relative quality. An interesting approach here, that will be explored throughout this project is that of coevolution between the test suite and the candidate patches - fixes should evolve to pass the tests while tests evolve to discriminate fixes [11]. The full test suite would then likely be reserved to a final step when considering breaking out of the EA loop, when one or more fixes have been found with maximum fitness.

2.4 Automatic Program Repair

Automated Program Repair [22, 56] looks to minimize the efforts associated with correcting faults that have been identified in a program. The search procedure is guided by correctness criteria typically given as a test suite [56] containing a failing test case.

Fixes are not cast in stone, human interaction should be sought after for several reasons:

- act as a final check of its correctness - test suites used to evaluate fixes for minimally-complex programs are necessarily incomplete
- ensure conformance to stylistic guidelines (albeit linter should already be integrated into the framework's pipeline)
- potentially distinguish among several accepted fixes, differing in non-functional properties such as size, understandability, speed,...
- offer feedback to the repair process - help finetune parameters,...

Still, such a framework in the Continuous Integration (CI) pipeline, automatically suggesting fixes when committed code is found to be faulty (as discussed in [56]), would free up developers to spend more time on the functional aspects of the programs. Moreover, continuous vulnerability detection, and user-submitted reports (eg. appropriately-tagged GitHub/GitLab issues), could trigger automated program repair. This would ensure some end-of-life support for the program even when the team migrates to another project.

This holistic approach to program repair, is unlike most implementations thus far which focus solely on program repair - faults are handpicked even when applied to real projects, by looking at previous manual fixes. Repairnator[12] does monitor CI for test failures, and as such operates on active open source projects with a test suite and CI. In our project we can relax these requirements resorting to automatic test generation. While admittedly lacking in effectiveness, its account helps pave the way for *"true industrial application of program repair"* [12] which we will be striving for.

Implementations

Perhaps the most famous example is the GP-based GenProg [57], with its representation based on list of edits operating over statements, and limiting edits to statements that were exercised by bug-revealing (negative) tests which are prioritized by frequency. It revealed great results in fixing 55 out of 105 bugs in large-scale open-source programs [3]. The validity of these results has, however, been disputed (cf. Sec.2.4).

GISMOE [48] proposes extending the source of fragments from which to perform *plastic surgery* (cut-paste) to related online repositories. Its application [55] to speeding up complex software (50k LOC) was met with great results (70x speedup). There, modifications were targeted at heavily used code rather than the location of fault.

Meanwhile [58] had a similar representation except its edits could affect individual tokens in the statement (not treaten as a whole). Furthermore, it is integrated into a project in-development for online improvement, having identified and corrected 22 bugs, with the GI process being run at night-time on bugs found during the day.

And still we have [59] which operated over the AST, albeit limited to small test cases and supporting *only* a subset of Java. The author's earlier joint work [60] appears to have introduced the idea of coevolution of test cases to program repair - in the hopes of creating an *arms race* between tests that improve at finding bugs and candidate fixes that eventually reach a correct fix.

PAR [4] relies on fix patterns synthesized into ten parameterizable fix templates - eg. the variable that is null checked. These templates are then used to rewrite the program's Abstract Syntax Tree (AST). Negative test cases are exercised to locate the likely defective statements, on which the evolutionary algorithm attempts to apply the templates using extracted AST nodes as parameters.

While the majority of implementations, 96% according to a 2017 survey[8], are based on GP (the generate-and-validate approach), other approaches include:

- SemFix [61] is based on program synthesis - the construction of programs from a specification [8] - where repair constraints are extracted via symbolic execution. This process is applied to a sorted list of suspicious statements, whose extracted constraints can can be solved with SMT to generate code fragment fixes. Limitations include the fact that repairs operate on a 'single-statement', and that symbolic execution is exhaustive, thus suffering when scaled to complex programs.
- DeepFix [62] relies on deep learning to fix compiler errors, usually syntactic issues such as missing tokens - more specifically through recurrent neural networks with attention weights assigned to each program token, based on error information (namely reported line(s)). These fixes are significantly easier to check as compilation is just a binary success function - if multiple errors then it is iteratively applied. This approach entails the difficulty of understanding exactly how these fixes are generated - some black-box elements. With an EA, despite the inherent randomness, we can trace how different individuals are being generated. Machine learning based approaches thus seem to be more appropriate to fixing compiler errors, and perhaps associated with a static analysis tools, where the messages can be inspected.
- SapFix [63] is an end-to-end industrial framework at Facebook focusing on the correction of null-deference faults, based on patterns inferred from previous fixes. It integrates automated testing for the detection and localization of faults, and requires a final step of human validation.

Discussion

Maximizing the guarantees that a proposed fix is correct is of paramount importance. In [64], patches from generate-and-validate systems, with one being GenProg[57], are analyzed and found to be overwhelmingly incorrect, some even introducing vulnerabilities.

Just the job of evaluating if a test case passes is complex and problem specific - much more may be outputted from execution that just than the returned value or what is written to the standard output. This is one of the areas that [64] found existing approaches to be lacking - some tests would only check the exit code and not the correct output (*weak proxies*). Furthermore, improving the test suites and methods did not produce better results. The authors proposed two explanations: the search space (constrained by the representation) does not contain the solution, or the fitness function gradient is not smooth, from the use of a single negative test case that distinguishes the fix from the original.

Still regarding a test's possible outcome, we would ideally be able to determine not just whether a test case failed, but also by how much, so as to integrate it into the fitness function and smooth the search space [59].

The 2019 survey [56] identifies three core challenges for program repair:

- ensuring the quality of repairs - will be tackled through dynamic fitness functions where the subset of tests evolves alongside the candidate fixes, converging towards the full test suite, which is not limited to tests available in the original project.
- extending the scope of addressable problems - we do this first of all by ensuring that fixes can be represented, by allowing the program AST to be manipulated according to C's grammar. Since this significantly increases the search space, the repair procedure will be restricted to code excerpts likely to be the root of the vulnerability. Moreover, knowledge of the vulnerability will be employed, by seeding the grammar with parameterizable fix patterns (that should be easily agumented).
- integrating it into the development process - our approach is mainly through continuous vulnerability detection (which with fuzzing has the side-effect of increasing the test suite), and developer interaction (for accepting and/or giving feedback).

Chapter 3

Framework for Automated Vulnerability Correction

The proposed framework for end-to-end vulnerability repair is laid out in Fig.3.1:

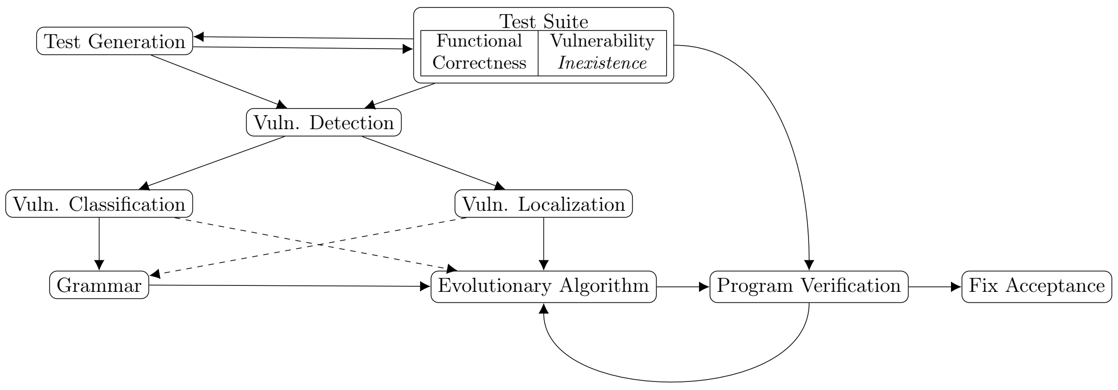


Figure 3.1: Proposed Framework for GP-based Vulnerability Correction.

Given a program, it begins by compiling a **test suite** through automatic **test generators**, that will help the EA's fitness function check that the functional correctness has been preserved.

Vulnerability detection is the trigger for the repair process. We initially limit the framework to repairing memory safety-related vulnerabilities in C programs, and as such look to couple the test generators with AddressSanitizer (ASan).

The repair process is based on GP. An important consideration is that the candidate fixes, once plugged into the original code, should compile. Syntactic correctness is achieved through the use of the language's **grammars**, while enforcing typing correctness remove a significant chunk of semantic errors.

Vulnerability localization is then used to limit the search process to specific code snippets, with the grammar being seeded with (and limited to) identifiers in their scope.

Vulnerability classification is used to identify fix patterns with which the grammar is seeded, consisting of parametrizable templates added to its productions (eg. bound-checking `if(<lval> > <const>-1){ ... }`). Furthermore, the EA will consider specific variation operators aimed at the efficient application of these patterns (eg. wrapping [13] a set of statements in an `if` construct).

Repair then proceeds, with the Evolutionary Algorithm **generating** candidate fixes that are **evaluated** to check that the vulnerability in question has indeed been fixed, without introducing other vulnerabilities or affecting the original program's functional correctness.

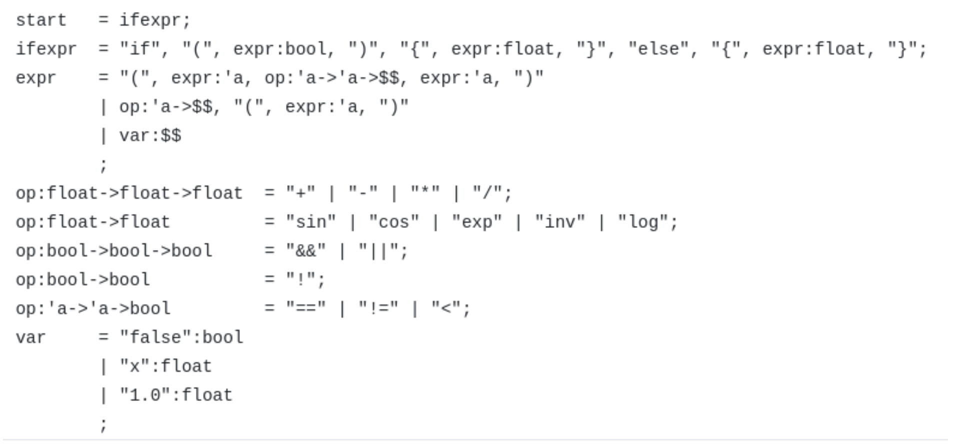
Finally, if a fix is found by the EA, it is proposed to the developers for **acceptance**/feedback.

The remainder of this chapter presents preliminary work and open ideas for the generation (dealing with the **search space**) and evaluation (defining efficient yet trustworthy **fitness functions**) of candidate fixes, the focus of our implementation efforts.

3.1 Search Space

Grammars are used for ensuring the syntactic correctness of the individuals generated, being specialized even further through seeding. In [65] the authors apply GE to the automatic generation of test data, wherein the grammar is seeded with literal values extracted from the program in question.

Preliminary work focused on problem-independent typing with support for polymorphic functions, common in operations like equality. This implementation works given any such type-annotated grammar as illustrated in Fig.3.2, and accompanied by an evaluator of candidate individuals. We will then narrow in for the C language, through support of its additional constructs like pointers, arrays and structures.



```
start   = ifexpr;
ifexpr  = "if", "(", expr:bool, ")", "{", expr:float, "}", "else", "{", expr:float, "}";
expr    = "(", expr:'a, op:'a->'a->$$, expr:'a, ")"
        | op:'a->$$, "(", expr:'a, ")"
        | var:$$
        ;
op:float->float->float  = "+" | "-" | "*" | "/";
op:float->float         = "sin" | "cos" | "exp" | "inv" | "log";
op:bool->bool->bool     = "&&" | "||";
op:bool->bool           = "!";
op:'a->'a->bool         = "==" | "!=" | "<";
var     = "false":bool
        | "x":float
        | "1.0":float
        ;
```

Figure 3.2: Type-annotated grammar for the split problem

Type annotations consist of concrete types (here, `float`s and `bool`s), polymorphic types in an ML-style (begin with a single quote), functions in a curried representation (`->` separated, with the last being the return type), and $$ being a syntactic convenience for inheriting the production's left-hand-side (LHS) symbol's type. Additional extensions include supporting comments for documentation, and reducing verbosity by having the RHS inherit the type specified in the LHS (as in the various `op` productions).

Enforcing typing correctness has an important drawback: we need to ensure the existence of a production fulfilling every possible LHS's type constraint. For example, the `!` operator must exist in Fig.3.2's grammar, for when a bool-typed `exp` selects its 2nd production rule. An alternative would be *looking ahead*, instead of considering only the current context, although this only done to constraint a polymorphic function's further arguments based on its first.

For crossover to maintain syntactic and typing correctness, the selected subtrees' roots will correspond to the same nonterminal and type. Mutation amounts to generating a subtree respecting its root node's typing constraints.

Regarding bloat control, mutation stops considering recursive productions once the specified depth limit has been reached, by tracking which nonterminals have already been expanded. Meanwhile, crossover needs to consider the depth of the selected subtrees, with the straightforward solution is to consider the offspring generated by replacing the lower-depthed subtree.,

3.2 Fitness

The fitness function is essential. Each candidate fix must be run on a corpus of tests to ensure its correctness. And for anything other than toy programs it will be impossible to attain full coverage - besides the fact that solutions could alter the original program's control flow for which tests were generated. We will therefore need to consider a subset of tests that maximizes our trust that correctness is preserved, and consider the possibility for a fix to overfit on these - set aside a final testing set (possibly just the remainder of the test suite).

Here, we can integrate the concept of staged fitness function [41]. Consider teaching a robot how to play soccer, instead of *telling* it straightaway to get a positive goal difference, we should start by simply getting it close to the ball, and progress from there. For functional correctness we can slightly adapt this: tests are divided into stages, and an individual is only run on the next stage if it passed the current [41].

Alternatively we would increase the small subset of tests as the populations' overall fitness increases (dynamic fitness function). That is, early on high-coverage testing is likely to be *overkill*, while later it helps us avoid overfitting. An interesting concept here is that of coevolution, where distinct populations cause each other to adapt, which can be applied to evolving the subset of tests used for fitness in conjunction with the program modifications [60] (2-Population Competitive Coevolution in [40]). Test cases would be assigned a fitness score based on how many programs fail on it - this determines their ability to discriminate, and if too low should be discarded (could be because fix doesn't pass through where we are modifying, very specific edge case not likely to be altered, etc). Another important question is that each possible execution path should be exercised by at most one such test case - minimize redundancy.

There's an interesting parallel with online judges for programming problems, which have the similar goal of assessing a submission's correctness with a reduced test suite (offer quick feedback, serve all participants,...). Codeforces [1], in particular, encourages contestants to find failing test cases for others' submissions (known as *hacking*), much like we hope to find test cases that discriminate candidate fixes' fitness.

And even still, the fitness function is extremely expensive. GenProg on a set of benchmark problems found that 64% of the time was spent running test cases [9]. And as much as fault localization speeds up the search procedure, fitness will run on the entire program. Fortunately we can easily parallelize fitness evaluation of different individuals. Other slight optimizations include avoid re-evaluating individuals preserved through elitism, unless the fitness function has since changed (further yet, this information could be timestamped so as to only run on newly-added test cases).

[1]`https://codeforces.com/`

Note that even with grammar and typing, invalid programs can still arise namely through the introduction of infinite loops - these are identified by **timing out** the execution and should be given bad fitness since reverting/fixing it it may be fruitlessly complicated. Further, execution should be **sandboxed** to prevent unknown side effects from the arbitrary code that will be run, including limiting memory consumption.

Since we are fixing vulnerabilities, the fitness function is even more complex: we must consider the preservation of functional correctness and security while ensuring that the vulnerability in question has been fixed. There are thus two factors weighing into the multi-objective fitness - security and correctness - which will need to be balanced. Our goal here is to satisfy both criteria, with a straightforward approach being to create a single metric with the sum of successful test cases (wherein a solution would be accepted if everyone of those had passed). However it would be enticing to give a higher weight to the vulnerability-revealing test cases. GenProg [57] in particular assigns double the weight to *negative* (bug-revealing) test cases.

And then, even if maximum fitness isn't achieved, possible correctness issues would likely be due to extraneous modifications that could be left for a human programmer to handle. Alternatively, the hill-climbing cleanup approach described in [48] could be adapted to tree-based representations - we would keep track of which nodes had been modified and iteratively try restoring them to the original subtrees.

We also need to pay attention to *high* correctness fixes that just amount to keeping the original program as was (reverting initial mutations we introduce when initializing the population). This is an important point in code correction - the fitness *will likely get worse before it gets better* - eg. if we're adding a missing if, it could initially be too restrictive.

It could also be interesting to keep a Pareto front where the other *axis* accounts for bloat (newly introduced nodes/model complexity). [59, 60] account the number of nodes in the fitness function for controlling bloat.

Since the population is initially extremely similar, crossover restricted to operating on equally-typed nodes is likely to have little impact. As such, we could also explore scheduling the probability of applying mutation (adaptive mutation rate [40]), giving it a higher percentage earlier on.

Chapter 4

Genetic Programming for Vulnerability Repair Engine

In this chapter we describe the GP engine implemented for vulnerability repair according to the previous chapter's framework, tentatively named GPVE - **G**enetic **P**rogramming for **V**ulnerability repair **E**ngine. GPVE focuses on the generation and validation of candidate fixes, guided by other tools' reports of the vulnerability - namely for its classification and localization.

4.1 Implementation Details

GPVE has been developed in OCaml, an efficient functional programming language [66], widely adopted for program analysis due to its idiomatic manipulation of algebraic data types (namely trees). We resort only to CIL [1] for parsing C programs and `domainslib` [2] for its parallel programming *primitives*.

The remainder of this section aims to give insight into how each *component* is implemented, in order to contextualize the experiments carried out and their results.

4.1.1 Grammar

The grammar used follows CIL's representation of parsed programs which, being tailored to program analysis, simplifies away many redundant constructs - e.g., considers a single form of looping `while(1)`. This translates into fewer grammar productions, which results in a reduced search space. The fixes end up bearing limited resemblance with the original program. Minimizing these differences, by transplanting the fix back into the original source code, is certainly a necessity if the tool is to have widespread usage.

There is however, a conversion step aimed at reintroducing tokens that had been abstracted during parsing, so that the genotype representation corresponds to the derivation tree of the phenotype - allowing for efficient and cachable mapping. Punctuation like braces and semicolons are reintroduced, as well as parentheses for expressions whose precedence is being encoded through the tree representation. This was particularly troublesome for types

[1]Specifically Goblint's (`https://goblint.in.tum.de/home`) fork which adds support for recent OCaml versions and C standards `https://github.com/goblint/cil`.

[2]`https://github.com/ocaml-multicore/domainslib`

- whereas CIL's representation of a pointer to array of integers is `TPointer(TArray(TInt))`, the derivation must actually follow the *inverse* order:

```
Type -> TInt, TArray;
TArray -> "(", TPtr, ")", "[" exp "]";
TPtr -> "*"
```

The grammar was also rewritten to replace certain nonterminals by their only production rule, aiming to reduce the size of the search space. But these *simplifications* cannot be done indiscriminately, or we would impair the level of granularity at which the crossover operator can operate. Consider a grammar with a single nonterminal that enumerated all possible programs, crossover would in fact just amount to random search. On the other hand, introducing a nonterminal `branch` into our grammar of C would allow crossover to more frequently select the branches of two `if`s as crossover points, over other kinds of statement lists (e.g., a loop's body) they are an alias to.

We do not yet support all of the C language's constructs, as specified in the C99 standard we aim for. Labels, for instance, are tricky since they can only be attached once to a statement, and any further attempt results in a compilation error. There is currently no tracking of their usage to restrict their application, and so `goto` statements are not manipulated (although they may be present in the program under repair). Switch cases are also not supported, given their complex semantics and not being used in the programs targetted for repair.

Typing

We look to guarantee the preservation of correct typing within the generated fixes, although full support for C's type system is still not supported.

The current implementation looks at whether a certain expected type (of the genotype node being expanded) can be filled by any of the variables or calls to functions we have access to, through a set of *reductions* that mirror grammar productions (eg. dereference a pointer to go from `Pointer(Int)` to `Int`). Since a structure can have multiple fields matching the expected type, a tree of reductions is calculated that can then be traversed incrementally as productions are applied.

Polymorphism as required by operators such as equality is implemented. Variadic functions, however, require restricting the number and types of subsequent arguments based on the first's value (the format string), and so even though `printf`/`scanf` are used, their typing is not yet supported. There is also no support for unions due to the issue of tracking which of its members is active.

In case of unexpected/unsupported reduction, we fall back into untyped GGGP through random selection among all of the nonterminal's expansions. This node would then with a type `Undefined`, matching with all nodes of the same symbol.

Seeding

Typing is added by annotating the grammar, although the variables, constants, functions, and their types, are dependent on the program we intend to correct. The seeding step looks to address this, automatically filling in placeholders present in the grammar's file.

By default the grammar is seeded with all global declarations, as well as the local variables for the function to be corrected. Support for multiple functions' local variables, where the corresponding grammar expansions would be filtered according to the current node's scope is not yet fully supported, as vulnerabilities considered thus far require modifications to a single function.

The seeded grammar's file is then parsed (through `ocamllex` and `ocamlyacc`), where types correspond to a simplified version of CIL's representation, used for type checking.

Fix Patterns

The grammar is seedable with additional productions encoding common fix patterns for the vulnerability being corrected. The goal is to introduce domain knowledge to guide the mutation's generation of subtrees towards those most likely to lead to a fix. These are selected based on the vulnerability's type, classified through ASan's reports.

For example, if a program is dereferencing a `NULL` pointer, then an additional production for the `expr`ession nonterminal could be: `lval "==" "0"`. The nonterminal `exp` has nine expansions, one being a binary operation. There are 18 binary operators and its operands are also expressions, where one should expand to lval and the other to a constant - of which two are seeded by default: `"1"` for boundary values and `"0"` for NULL checking. As such by adding the pattern's expansion, our odds of generating such a partial subtree go from 1/26244 ($9 \cdot 18 \cdot 9 \cdot 9 \cdot 2$) to 1/10.

Additionally, although we consider low mutation rates, such patterns do not have to be generated with the correct nonterminal expansions, or at the correct tree location, since this can the be done by repeated applications of the crossover operator.

We aim to add further variation operators that emulate common fix patterns. A simple example is *wrapping* a subtree with a missing construct, or *lifting* an extraneous construct [13]. Generating these effects through subtree mutation/crossover would be extremely unlikely. As GPVE starts considering more than the current node's context, for its typing and generation, patterns for the application of these specific operators would be added.

4.1.2 Extraction

GPVE takes intervals of source code line numbers and restricts further operations to the corresponding subtrees. This fault localization is to be provided by some other tool, which in the case of this work was ASan through the stack traces in its error messages.

This so called *code extraction*, while optional (as otherwise the entire program is considered), is essential to focus the search on the places where the fault is most likely to be. Notice that the implementation *simply* consists of annotating tree nodes with an `edit_prop`, set to `NONE` if that node and its whole subtree can be skipped altogether during traversal - be it for applying a variation operator or for generating the phenotype which is cached. This approach is highly flexible, but particularly useful for adding support for dynamic localization of vulnerabilities, wherein the interval(s) considered are also evolved.

Further, as tree nodes are immutable objects, those that are not *extracted* for repair can be shared among the entire population without loss of performance or correctness. This allows GPVE's memory usage to scale gracefully with the population size [3] - as we look

[3] Although no appropriate method of measuring was found, other than monitoring the process's fluctuating resident memory usage in `htop`.

to extract only small fractions of code.

4.1.3 Initialization

For our project, the typical approach to initializing the population, generating individuals from scratch, is not adequate since we already have a single established starting point, the original program. However, cloning it for the initial population would lead to no diversity in the initial population.The solution adopted was that of seeding the initial population with small changes to the original program through the application of the mutation operator.

4.1.4 Variation

The only variation operators currently supported are subtree mutation and crossover, respecting typing and syntactic correctness.

Since the genotypes are fairly unbalanced trees, with lists being enumerated one element at a time through `<list> = <element>, <list>` productions, the `depth_growth` parameter was introduced to bound mutation's subtree generation. The idea being that the depth limit at which recursive productions are restricted should take into account the maximum depth of the original subtree, rather than the whole tree's. As such, each subtree is allowed to grow by the same margin, defined by the parameter `depth_growth`.

4.1.5 Fitness Evaluation

GPVE saves the phenotype[4] to a file, compiles it, and executes it with each case in the test suite. This process is immediately cut short in case of compilation error, or execution timeout for some test - thus considering these individuals invalid (fitness of -1). If run to completion, the fitness score amounts to the number of test cases that ran successfully - no vulnerability reported by ASan and where the expected output was matched.

The expected output is known for each test case as we start out with the *secure* [5] program, and only then inject it with vulnerabilities. In a real use case scenario we would not know the expected output for the vulnerability revealing test case, so any program that exited successfully would pass it - the expected output would be a placeholder that matched all, leaving it to the developer to decide if the desired functionality was preserved and adjust accordingly.

The duration before timeout should be specified with consideration for the program being fixed, as the idea is to detect the introduction of some infinite loop which is not an issue we are proposing to correct[6]. This assumes there were no infinite loops in the original program, or that they are not exercised by any case in the suite.

Since ASan's instrumentation turns the executable into well over 1MB, these are deleted upon fitness completion to minimize disk space requirements. Further, if not debugging then the phenotypes for intermediate generations are deleted (several terabytes of data

[4]Generated by collecting the genotype's leaf nodes. Additionally, it is processed by a primitive code formatter for legibility.

[5]Input size needs to be limited, but we are not targetting their parsing.

[6]Because how do you fix an infinite loop, if removing it altogether makes the fitness go from -1 to 0 while losing functionality.

have been written - checked in the disk's NVMe SMART log, making the support for interpreted languages an appealing proposition).

Dynamic Fitness Function

Given the reduced size and execution time of the test suites considered thus far, they are entirely used to assess each individual. However, we still experiment with dynamically attributing weights to each test case, based on their perceived difficulty.

Let n be the number of test cases and $weights = \begin{bmatrix} w_1 & \cdots & w_n \end{bmatrix}$ be a vector where w_i is the number of individuals who have failed to pass test t_i. The fitness score for an individual, where the vector $passed = \begin{bmatrix} p_1 & \cdots & p_n \end{bmatrix}$ encodes with a $\{0, 1\}$ whether it passed test t_i, is given by the formula:

$$\frac{passed \cdot weights}{\sum weights} * n \tag{4.1}$$

The final multiplication by the number of test cases aims to preserve compatibility with existing acceptance criteria, seeing as that is the maximum fitness we can get with the *static* fitness function that counts the number of passing tests.

The weights are updated at the end of each generation, and initialized with ones making it equivalent to the *static* fitness function for the first generation . This implies that although we still only run individuals once through the test suite, even if preserved through elitism, the associated score must be recomputed each generation from the cached *passed* vector.

Higher scores are therefore given to individuals that pass *hard* test cases in which the majority of individuals fail, even if perhaps failing on most others. This way we can promote population diversity and avoid early convergence. Another benefit is that we get a larger domain of values for the fitness function, because we now have up to 2^n possible values rather than only $\{0, \cdots, n\}$ values.

Here, convergence is actually represented by a decreasing average/best fitness, as the test cases the population is failing on get increasingly higher weights.

Ideally the repair scenario (program, its vulnerability and its test suite) would allow for some candidate fix to pass only a few tests, but still be promoted through selection as the remainder population had not been able to pass those (although passing a larger number in total). While this will probably not help for the programs currently being considered, it is something we will look to build upon in the future, namely in identifying additional heuristics for updating these weights.

4.1.6 Selection

The same comparison function is used for ranking individuals according to their fitness: in tournament selection to return the best out of a random sample, and for elitism to preserve the first N individuals out of the sorted population.

Here, we experimented with considering not only the score each got from running on the test suite, but also the size of the genotype's tree, to encode our preference for *simpler* fixes, and so that a single fix can be proposed in case of multiple acceptable solutions.

Preliminary experiments showed the population would often converge towards the preservation of equal-fitness individuals with slight functionally-neutral variations that bypassed the removal of duplicates described below (Sec.4.1.6). The solution adopted was that of only **considering the genotype's size when comparing acceptable fixes**, which also serves as an implicit stage of fix *reduction* (and until then looks to prevent early convergence). This, however, can also lead to significant rewriting the original program if it - counter-productive to our goal of easily understandable fixes. A more adequate metric would be the editing distance between the original program's genotype and the fix's [7], but this is left for future work (Sec.7).

Elitism

Elitism, while beneficial to ensure the fittest individuals are preserved through generations, can also force population convergence if the individuals preserved lack any diversity.

To address this, preliminary experiments were conducted with a `maxdups` function, that allows the preservation of a certain number of *duplicate* individuals. The identification of such *duplicates* is actually based on the comparison function used for selection, considering their fitness score, and size only if an acceptable solution is being assessed - otherwise it would promote small functionally redundant variations of the same individual.

The number of *duplicates* allowed is currently expressed as a fraction of the individuals preserved through elitism, and should take into account the possible values the individuals' fitness may take. Specifically, it should not be such a high value that would lead to the preservation of invalid individuals.

4.1.7 Acceptance

An acceptable solution is a a candidate fix that successfully passed all tests in the suite, found within the maximum number of generations (`num_gens`) allotted, or optionally within a deadline (user wants feedback within few minutes, or limited resource usage when automatically triggered).

Since we may tend to find solutions faster for some vulnerabilities, we can specify the number of generations over which to improve an accepted solution with respect to the additional selection criteria (namely its genotype size) before early termination of the evolutionary process.

In any case, the individual with highest fitness is fully logged to the `"num_gens+1/"` folder, with its phenotype (C code) and genotype (internal tree representation), and is the fix proposed by GPVE.

Fuzzing

Ideally the proposed fix would be subjected to automated test generation, to identify paths that may not have been covered by the existing test suite.

Each program has a fuzzing target for use with LibFuzzer, that would replace their insecure input parsing, but it required dealing with missing/conflicting declarations and was not implemented in time for the experiments.

[7] If the vulnerability was injected, we can even compare against the *desired* fix.

4.2 Reproducibility and Replicability

When arguing for the benefits of GPVE's approach, reproducibility of results is a necessity to reach scientific consensus [67].

Although to show this we are lacking comparison with existing tools, efforts were nevertheless made to detail the implementation, experiments and its results. Further, GPVE's runs should be replicable if in possession of the repository from which they were run. To achieve this:

- The random number generator seed is logged, and optionally taken as a command-line argument - else a random one is selected from `/dev/urandom` - Ocaml's `Random.self_init`).
- The data of the latest commit is logged, with GPVE checking `git`'s working tree that no implementation file has been edited (and *refusing* to run otherwise). Here, we exclude `engine.ml`, as this file is saved to the root of the run's directory, and contains all of the parameterization - thus allowing for multiple batches of runs in the same commit.

Scripts for generating graphs for any batch of runs are also provided. These can be extracted from `.zip` files as the repetition of a lot of code among the phenotypes (that which is not extracted) leads to great compression rates.

4.3 Performance

EAs are (embarassingly) parallelizable - fitness evaluations for each candidate solution in the population can be considered independently, and so can the generation of new individuals.

Here we take advantage of the recent efforts in Multicore Ocaml [68], which supports shared-memory parallel execution through *domains*, each running as a separate system thread [69]. Domainslib [8], allows us to divide the workload in a simple manner through the the creation of domain pools, and the `parallel_for` primitive allocates chunks of a loop's iterations amongst a pool's domains.

By default GPVE uses all physical cores, ie. half of `nprocs`' output assuming a processor with hyperthreading. If the number of cores is set to `1` then no domain pool is set up and it runs as if in *Singlecore* OCaml. To get a sense of the speedup obtainable we ran the same workload (5 generations of 1000 individuals on stackbufferoverflow) with the configuration outlined in Sec.5.1. The results are presented in Table 4.1[9]:

These results obviously are obviously highly dependent on the program under repair and the size of the extracted code, but are a good indicator for estimating execution time of the results reported in Ch.6 (cf. Sec.6.2).

Also note that not the entirety of the program is parallelized, such as the initialization of the next generation's population by sorting the old one according to the fitness scores.

One final important concern: as it stands the program uses a global random number generator, meaning that locks are required to access it and calls can be interleaved. This

[8]`https://github.com/ocaml-multicore/domainslib`

[9] In the style of `https://github.com/ocaml-multicore/parallel-programming-in-multicore-ocaml`.

Cores	Time(s)	Speedup
1	498.783	1
2	251.161	1.9859
4	129.273	3.8583
8	69.235	7.2042
16	42.680	11.6866

Table 4.1: GPVE multicore speedup

can have quite a large impact in performance, however profiling GPVE with `perf` shows that fitness evaluations (compilation and test execution) takes up most of the execution time [10]. But it also jeopardizes the reproducibility of results since these calls can be interleaved. Each domain could have its own generator, but the allocation of individuals to domains is still non-deterministic by default - this can however be hardcoded at a slight hit to performance, as individuals can take vastly different times to generate/evaluate and so the load could not be spread out on the fly.

4.4 Verification

A wide array of techniques were used to verify GPVE's implementation:

- Dynamic invariant checking through assertions, eg. after running the mutation operator check that only one subtree was altered.
- *Incremental integration* testing for each of the program's module, through their own *main* function exercising the functionality to be exposed, ensuring they properly build upon their dependencies
- Regression testing to ensure previous assumptions regarding the type system are preserved as we build upon it.
- Property checking in the style of QuickCheck [70] was used, through the `qcheck` package, to experiment with generators for providing constrained random inputs to the `maxdups` function (Sec.4.1.6)
- OCaml's own typechecker is used for building upon the parser combinator used for the vulnerability locations - since the main difficulty is ensuring each component remains composable, ie. types match, the actual parser definition shadows a sequence of others solely there to raise compilation errors. The actual correctness of the parser is debugged through its result - which in case of failure specifies the missing token in the given context.
- Sanity testing, when ensuring that the converted program matches the grammar, ie. every node's leaves match one of the grammar's expansions.

[10]Different levels of optimization were tested for the compilation with no noticeable speedup (compile speed traded for code speed evens out).

4.5 Usage

As it stands, GPVE is expecting to have a set of injected vulnerabilities, each in a separate version of the program.

To add support for a program, GPVE expects there to be a corresponding directory under "`programs/`", as illustrated below:

Listing 4.1: Expected file structure of *program* to be repaired by GPVE

```
programs/trie
├── patterns
│   ├── nullpointer.exp
│   ├── overflow.exp
│   └── useafterfreee.instr
├── tests
│   ├── 1.in
│   ├── 1.out
│   ├── 2.in
│   ├── 2.out
│   └── ...
├── runs
│   ├── pop100patterns
│   │   ├── nullpointer
│   │   └── ...
│   └── ...
├── trie.c
└── vulns
    ├── locs
    ├── doublefree.c
    ├── nullpointer.c
    ├── overflow.c
    ├── useafterfree.c
    └── useafterfreee.c
```

Here, it will look for the test cases under "`tests/`", with the expected inputs ending in "`.in`" and outputs in "`.out`" (and with matching basenames).

Vulnerabilities to be fixed are placed under "`vulns/`", looking for "`.c`" files that it will then preprocess with "`clang -E`", and whose basenames will act as identifiers thereafter. Currently single file programs are expected[11], that are parsed by CIL and converted to our representation, with code extraction following the lines optionally specified in the `locs` file - in the format "`<vuln> <func> [<start> <end>]`"[12].

Fix patterns are optionally provided under "`patterns/`", in files "`<vuln>.<nt>`" corresponding to grammar expansions added to the nonterminal `nt`.

A batch of runs is considered to be the sequential correction of a set of vulnerabilities (all, by default) for a certain number of runs, under the same parametrization, and with all output placed under the "`runs/<batch>/<vuln>/<timestamp>`" directory. Checking the modifications of a proposed fix can be done with "`git diff -no-index`" between its phenotype and the original vulnerable program. Passing this batch directory to `graphs.py`

[11]Not due to technical limitations, since CIL can merge multiple source files into a single parsed structure.
[12]`<func>` is the function whose local variables will seed the grammar.

will generate all graphs and tables presented in this document, by filtering its logs with Awk.

All parameters controlling the evolutionary process are specified in the `engine.ml` file, namely the population size (`pop_size`), maximum number of generations (`num_gens`), tournament size (`tsize`), number of individuals preserved through elitism (`elitism`), maximum number of *duplicate* individuals preserved (`maxdups`), probability of crossover and mutation (`prob_crossover` / `prob_mutation`), maximum tree depth growth (`depth_growth`), whether to use the dynamic fitness function (`dynamic_fitness`), the acceptance criteria (`deadline`, `gens_to_improve`) and timeout for program execution (`timeout`)).

Chapter 5

Experimental Setup

In this chapter we introduce the experiments looking to evaluate GPVE's ability to fix vulnerabilities, and study the impact their type, location and associated fix patterns had. Sets of vulnerabilities were injected into two data structure implementations, for which test suites were generated to evaluate candidate fixes on.

5.1 Configuration

Experiments were carried out on a 16-core CPU machine (AMD 5950X), with 16GB of RAM and an NVMe SSD [1]. GPVE was compiled with version `4.12.0+domains` of the OCaml compiler (variant with multicore support), and uses `clang 12.0.1` for compiling the candidate fixes with ASan.

5.2 Programs

The selection of programs on which to assess GPVE is problematic - we do not want trivial programs as they offer no functionality to guide the evolutionary process, but large-scale programs are ill-suited for time constrained experimentation (ideally we would run those until it found a fix, not 30 times over).

After looking through several repositories (ManyBugs [72], SIR [73], afl[2]/LibFuzzer[3]'s trophy cases) the conclusion was reached to inject vulnerabilities into implementations of a `skiplist` and `trie` data structures, wherein a simple command parser reads sequence of operations from `stdin`, to facilitate fitness testing.

These are programs that provide concrete functionality through an abundant usage of pointers and buffers. And while this is familiar code to us, it was implemented in years past with no intentions of running GPVE on them.

The source code for `skiplist` is provided in Appendix A.4, and B.4 for `trie`. Note that the goal is to fix vulnerabilities injected in the actual implementation of these data structures, not in their input parser where buffer overflows are actually trivial to trigger

[1]It did feel compelling to cite it as a co-author, considering it found the reported fixes, just as Doron Zeilberger does for his proof-finding computer [71].

[2]`https://lcamtuf.coredump.cx/afl/`

[3]`https://llvm.org/docs/LibFuzzer.html#trophies`

with malformed commands. Secure parsers were implemented for use as fuzzing targets, although without a grammar specification for the inputs, the majority of generated tests would *focus* on debugging the parser rather than the data structure.

5.3 Vulnerabilities

In this section we describe the vulnerabilities injected onto the `skiplist` and `trie` implementations - each resulting in their own version of the original program.

SARD's Juliet Test Suite contains test cases for evaluating static analysis tools (small programs whose only purpose is to contain a vulnerability), organized according to their CWEs. Some of these were adapted by looking for locations in the original program where it would make sense to have similar ones, in efforts to generate realistic, even if relatively simple vulnerabilities. These should, of course, be supported by ASan and thus related to memory safety, and should also look to exercise different paths of the program - to also be revealed by different test cases.

Below we give an overview of each vulnerability, and the idea behind the choice of location interval and fix patterns, which were manually selected but based on the the reports generated by ASan (listed in tables A.1 and B.1). Given the overlap between the two programs and their vulnerabilities, we will focus on the `skiplist` program, although the information is also summarized for `trie`'s in Appendix B's tables.

- ***Double Free*** (doublefree [4]): Adds a loop freeing nodes pointed at each level, which is unnecessary since every element is reachable through the first level (`next[0]`), thus leading to attempts to free the same node multiple times. However this is somewhat a misnomer, since the `NULL` check actually leads to a `SEGV` before we would attempt to double free. The expected solution is the removal of this loop construct, and we consider its lines as the interval for extraction.
- **Free Static Array** (freenondynamic): Nodes removed from the list are freed, alongside their buffer of pointers to others. Here, we add an attempt to also free a static array, looking for GPVE to remove it.
- **Heap-based Buffer Overflow** (heapbufferoverflow): Consists of an incorrect index initialization, with the caveat that we do not let GPVE consider the corresponding line, but rather the three incorrect usages that follow from it (and trigger ASan). An expression pattern was specified for decrementing a variable, and another interval was considered now adding the initialization line, given the difficulty of finding a fix.
- **Memory Leak** (memoryleak) Detected with ASan through the associated Leak-Sanitizer, this is the only vulnerability for which the stack trace cannot point to a location (detected at termination). Here we would need to consider all functions where memory is freed , in hopes of GPVE removing the leak while preserving functionality (several functions would eventually need to be considered). As such, the interval contains the entirety of Free function. The purpose of the alternative location was more so to highlight how, even though we restrict it to a single statement, GPVE can still add statements (by manipulating its `stmtlist` parent node). A fix pattern is also considered - a call to `free` with an `lval` as argument (as opposed to any expression).

[4]Links such as this one point to the `diff` file highlighting how the vulnerability was injected - in them, the function name points to the location in the original source code.

- **Null Pointer Dereference** (nullpointer): Uses a `do...while` so that the NULL check occurs only after the first iteration of the loop's body. Given CIL's representation of all looping constructs as an infinite loop, it should give rise to element fixes wherein its body and termination condition are swapped.
- **Null Pointer Dereference #2** (nullpointerr [5]): Removes a check for NULL pointer before dereference. Given GPVE's difficulty in finding a fix, we experiment with considering a larger interval so that recombination has access to more genetic material, since a similar check is performed in the loop preceding it. We use an `lval`'s NULL check (points to non-zero address) as a fix pattern.
- **Stack-based Buffer Overflow** (stackbufferoverflow): Simulates a programmer mixing up variables representing indexes. An interval is selected allowing for a fix to be reached either by correcting the wrong index's initialization or its usage. The alternative location is only the line reported by ASan - the incorrect usage, thus reducing the search space.
- **Use After Free** (useafterfree): Makes it so that when removing a node, not all nodes that previously pointed to it get updated. If an atempt is then made to traverse the list it will attempt to access the removed node. ASan reports the location where it had been freed, which we use for fault localization.

5.4 Test Suites

The test suites used for fitness evaluation of each candidate fix should have maximum feasible coverage, looking to guarantee the correctness of accepted fixes.

Knowledge of the underlying program was employed to generate the test suites, followed by checking `clang`'s SourceBasedCodeCoverage report of branch coverage as a sanity check. This tool also displays the source code highlighting the branch for which one/both of the boolean values was not exercised. Path coverage would have certainly been preferable, although it was not supported.

Each vulnerability injected was also ensured to be detected by one (or more) of these tests. Had fuzzing been employed for fix acceptance, any failing test case would then augment the test suite for subsequent runs.

In the end, `skiplist` had a test suite of 13 expected inputs/outputs, while `trie` had 9. These are very low sizes, but given the fitness function's impact on the repair process's execution time, we aimed for a minimal set of test cases.

5.5 Experiments

Experiments conducted look to check the engine's ability to find fixes, and assess the isolated impact of several assumptions:

1. **Baseline** - consider a reduced population size of 100 to look for evidences of the correct functioning of the evolutionary process - namely in reaching an acceptable

[5]The repeated 'r' at the end to get a unique id, rather than using a number suffix, stems from a limitation in the parser combinator used for the location specification file (implemented/used purely out of curiosity).

Population Size	100 or 1000
Tournament Size	3 or 5
Elitism	10% of population size
Maximum Duplicates Preserved	20% of elitism
Number of Generations	50, no early termination
Probability of Crossover Operator	0.9
Probability of Mutation Operator	0.1
Depth Growth	5
Fitness Execution Timeout	1 second

Table 5.1: Parameters shared among experiments

solution after N generations through successive applications of the variation operators, as opposed to *luck* from an initial mutation.

2. ***Large* Population** - study the impact of a larger population size (1000 individuals). This translates into a proportional increase in generations/evaluations of candidate fixes. We also adjust the size of the tournament used for selection to 5.
3. **Fix Patterns** - study the impact of encoding fix patterns as additional grammar productions to be used during mutation, whose rate remains unchanged. For general use there would be a list of patterns associated with each vulnerablity type.
4. **Alternative Locations** - study the impact of the fault localization, by exploring intervals of lines that should make the correction harder, if it had previously found a fix with ease (and *vice versa*). While more than one interval could have been specified, we looked to simulate their inference from ASan's output by considering intervals *surrounding* the location where the vulnerability was revealed.
5. **Dynamic Fitness** - study the impact of the dynamic fitness function described in Sec.4.1.5, namely with regards to its ability to promote population diversity and avoid converge.

Each experiment is conducted for all applicable vulnerabilities' of both programs (not all have an associated fix pattern or alternative location), through 30 runs of the evolutionary process. These consist of generating/evaluating 50 generations of the population with no acceptance criteria, for consistency in the generated graphs (particularly averages throughout generations).

Table 5.1 lists the common parameters among experiments, that are also shared by both programs given their similarities. Note the low mutation rate, which hinders the application of patterns, but looks to avoid the evolutionary process from degenerating into a random search.

5.6 Data Analysis

Log data from each run contains information about the population and its best individual for each generation. These are filtered by an `Awk` script, to make it directly readable by a

Python script into a `pandas` Dataframe, used to generate all graphs through the `seaborn` package.

Graphs for the `skiplist` and `trie` programs are placed under Appendix A and B respectively. These are then selectively interpreted in the results chapter (Ch.6). For a given experiment and vulnerability, we are particularly interested in:

- The number of accepted fixes over its 30 runs, i.e., the *success rate.*
- The population's average fitness throughout generations, useful to validate that the population is indeed being evolved, and checking how soon it appears to converge.
- The distribution of the generations at which an acceptable first was first generated, giving an idea of how *hard* it was for it to be generated, and for how many generations we should let it run without doing so.
- The evolution of the best candidate fix's fitness in each run.

5.7 Threats to Validity

The selected programs are small representations of the engine's scalability, however they do allows us to more accurately reason about its remaining aspects, and already provide a practical use case for such a tool - an aid to novice students (although the tool's usability and understandability is still lacking).

The representativeness of the vulnerabilities, since these are quite textbook examples that should be detected through minimal testing and with an evident correction. We are somewhat constrained by the underlying program. Adding vulnerabilities to where they make sense, and taking the leap of faith that similar ones could occur in large scale programs - with the difference being, that the path taken to triggering them would accept a much smaller set of values. Again, the target of this tool is for developers inexperienced in software security, for whom these vulnerability fixes should be much less evident.

The lack of fuzzing, running the proposed fix on a set of unseen test cases, is certainly a limitation of our work. Likewise for not considering subsets of a more extensive test suite, focusing on the function in which the vulnerability was injected, although test inputs combining the data structures' different operations makes this challenging to ensure. As it stands we have limited guarantees regarding the correctness of the fix. However we do have strong indications of the engine's ability to evolve programs, so as to pass test cases that otherwise revealed vulnerabilities. Also note that the expected outputs for the program paths that previously revealed the vulnerability are unknown to GPVE - here we do not have the oracle that was the original program - and so developer analysis would always be necessary, at least to ensure functional correctness.

This page is intentionally left blank.

Chapter 6

Results

In this chapter we analyse the results obtained when GPVE is used to correct a set of injected vulnerabilities. The experiments analysed in Section 6.1 aim to assess GPVE's performance, and the impact of larger populations, fix patterns, alternative locations and a *dynamic* fitness function, with regards to a common baseline. Section 6.2 then shows how these results can be used to estimate the expected cost of generating a fix, while Section 6.3 dives into these fixes, highlighting GP's ability to evolve programs through complex code manipulation, but also the insufficient correctness guarantees provided by the limited test suites used for fitness evaluation.

6.1 Experiments

We start out analysing the Baseline experiment's results, with which all subsequent runs will be compared. Results shown are for 30 runs on the given vulnerability, and all generated graphs are left in Appendix A.4 for the `skiplist` program, and B.4 for `trie`.

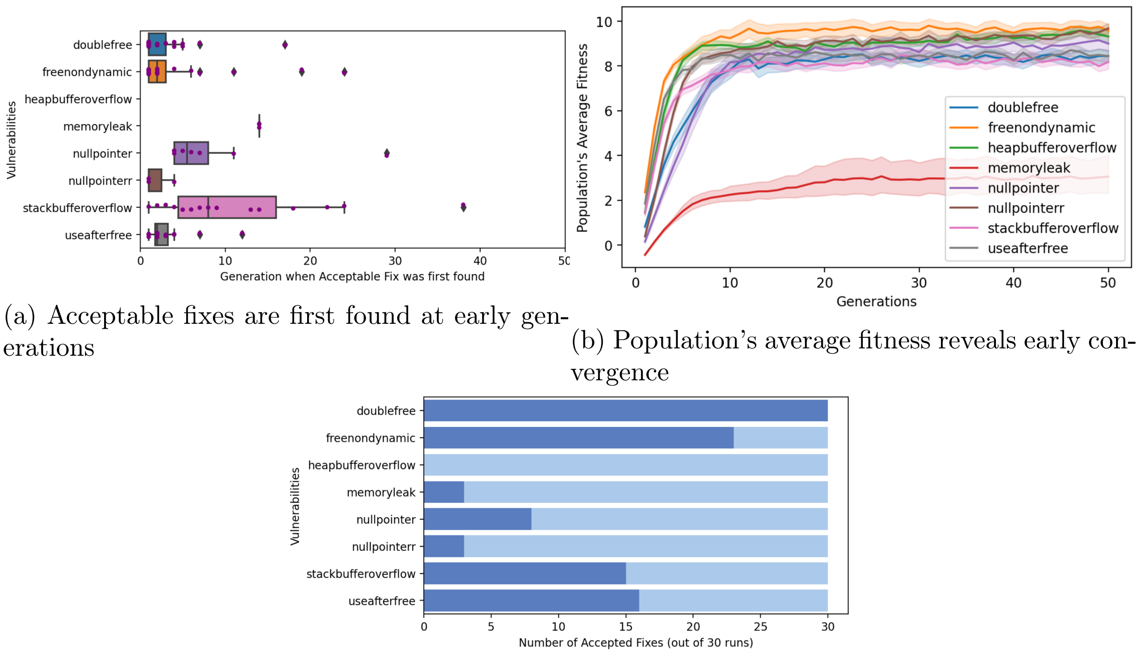
6.1.1 Baseline

The goal of the baseline experiments was to assess whether an acceptable solution was found with a limited population size of 100 - i.e., if evolution was able to find fixes.

Focusing on `skiplist`'s vulnerabilities, we do see that fixes tend to be found early on (Fig.6.1a), suggesting the importance of the population initialization's early mutation in providing *fresh genotypes* (new subtrees) from which to derive the fix. Some outliers do exist, with some fixes being found past the 20th and 30th generations.

Meanwhile the average fitness of the population increases rapidly in the first generations, stabilising around the 10th generation (Fig.6.1b), likely due to preserving the 10 fittest individuals through elitism, which might result in a loss of diversity.

Regarding the ability to generate fixes (Fig.6.1c) heapbufferoverflow is unsurprisingly hard to fix, with the default extraction interval requiring the modification of three incorrect index usages. doublefree on the other hand, removes the enclosing loop with ease, as constants added to the code fragment will eventually reach the statement incrementing the loop's counter, skipping it after just one iteration (cf. Sec.6.3.1).



(a) Acceptable fixes are first found at early generations

(b) Population's average fitness reveals early convergence

(c) Generates fixes for all but the (expectedly) hardest vulnerability, with modest success rates

Figure 6.1: "Baseline" experiment's results for `skiplist`

6.1.2 *Large* Population

A larger population shows a higher success rate (Fig.6.2c) and lower average generation at which fixes are found (Fig.6.2a) - expected given the proportional increase in number of candidate fixes generated/evaluated. However, GPVE still struggles on the heap buffer overflow vulnerability.

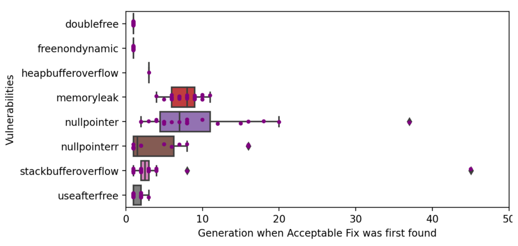
Analysis of tables A.4 and B.4 confirms fixes are indeed generated earlier on average, suggesting the number of generations could be cut down, or certainly bounded after successfully finding a fix - this was the approach initially adopted, however having each run potentially ending at different points would lead to misleading aggregate statistics.

Notice in Fig.6.2b how the average population fitness actually ends up decreasing when attempting to correct memoryleak. This certainly raised concerns, with analysis of a similar vulnerability's fixes in Sec.6.3.5 revealing how the inadequate support for memory leak detection allowed invalid individuals to pass all tests through chance. Since these are then preserved through elitism, they end up *polluting* the remainder of the population.

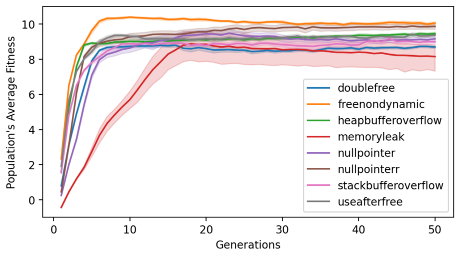
6.1.3 Fix Patterns

Fix patterns were used in the correction of a subset of vulnerabilities. Heap buffer overflow continued being unfixable with its restrictive interval of source code lines, while only a modest increase in success rate was observed for the remaining vulnerabilities.

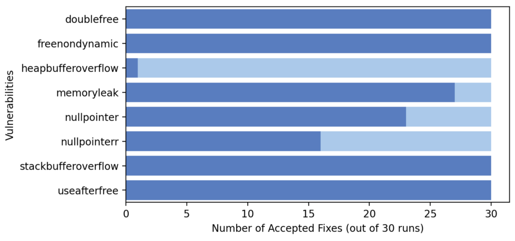
The lack of noticeable improvement should stem from the low population size (100) and mutation rate (0.1), which amounts to approximately 500 mutations over the 50 generations. Since the intervals of code GPVE operates still amount to hundreds of nodes selectable for mutation, of which only a few could generate expansions using the fix pattern, it is then up to crossover to appropriately fill the patterns' nonterminals.



(a) Vulnerabilities fixed at earlier generations, although after an increased number of evaluations

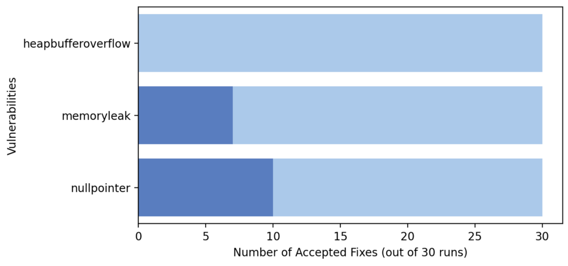


(b) Average population fitness for memoryleak worryingly plateaus and decreases



(c) Higher success rates

Figure 6.2: "*Large* Population" experiment's results for `skiplist`



(a) Modest increase in success rates, when compared to baseline results

Figure 6.3: "Fix Patterns" experiment's results for `skiplist`

Future experiments should look at larger population sizes, and potentially larger mutation rates (without letting the evolutionary process *degenerate* into random search).

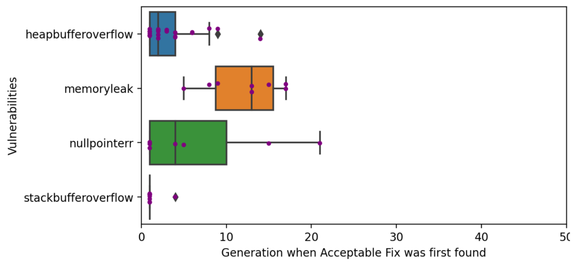
6.1.4 Alternative Locations

We evaluate the impact that the alternative locations given to GPVE had in its performance.

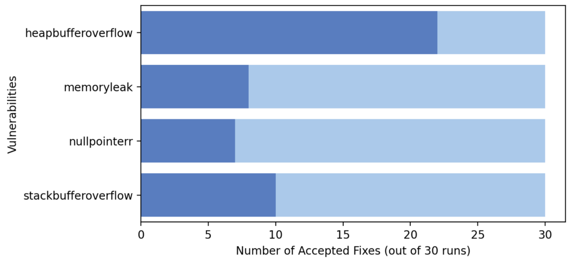
For instance, by increasing the interval of source code lines in heapbufferoverflow to allow for the index's correct initialization, GPVE is now able to frequently generate fixes in spite of the larger search space.

In what concerns nullpointerr, the success rate also increased with a larger interval of source code lines (Fig.6.4b). Since a similar check to the one that is missing became accessible, crossover can now *swap* it into place. However we also see the increased search space translated in fixes being found at later generations, on average (Fig.6.4a).

Despite now only operating over the incorrect line of code, the success rate on the stackbufferoverflow actually decreased. This might be explained by the fact that that GPVE was relying on crossover between other parts of the code to reach it, swapping the index usages. Mutation on the other hand, besides having a low rate (0.1), has depth growth set to 5 (Sec. 4.1.4). As such, if the index is ever selected for replacement through mutation, it is actually biased to generate complex array offsets since there is only one non-recursive expansion (eg. could use a node's `saldo` field).



(a) nullpointerr's increased search space affords it more material from which to generate the fix, although finding it later on



(b) Not all of the success rates matched our expectations - stackbufferoverflow's worsened with a more precise interval of source code lines

Figure 6.4: "Alternative Locations" experiment's results for `skiplist`

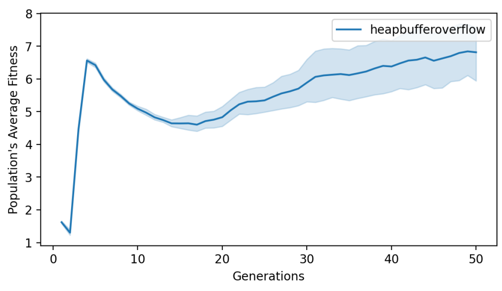
6.1.5 Dynamic Fitness

Analysing the results of the dynamic fitness function described in Sec.4.1.5 is challenging using the visualizations thus far considered. So while this is still very much ongoing work, with inconclusive results regarding its benefits, it did show promising results on what has been the hardest vulnerability to fix, heapbufferoverflow, fixing it twice on a set of preliminary tests.

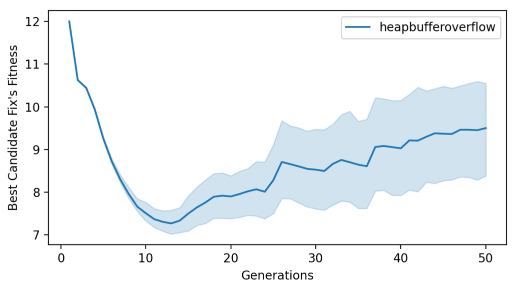
As such, we decided to run it with a population size of 1000, and it increased the number of times it found a fix to 12, as opposed to once when using the *static* fitness function. In Fig.6.5c we see how, although the best individuals' fitness starts out by decreasing, as the tests it fails on get increasingly higher weights, when it does pass them is accompanied by a sharp increase in fitness. Also notice how in Fig.6.5b the increase in the best individual's fitness is soon followed by the population fitness increasing (Fig.6.5a), as it gets selected for reproduction and spreads its improvement.

In this work all runs consisted of 50 generations, although results show they are often discovered early on or not all. Attributing dynamic weights to each test case shows its ability to detect fixes even at latter stages, by emphasizing candidate fixes that pass *hard* tests. One possible idea to explore further is the scheduling of the fitness function, initially static but in case of no noticeable improvement we would switch over to the dynamic version.

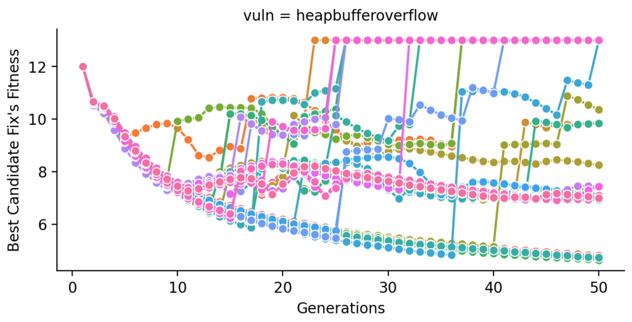
With this dynamic fitness function, it should also be useful to track the relative frequencies



(a) Average population fitness



(b) Average fitness of best individual



(c) Best individual's fitness for each run

Figure 6.5: Success on heapbufferoverflow with dynamic fitness function and population size of 1000

of scores in the population, and of how many new subsets of passing tests were obtained each generation, to get better insights into if/how population diversity is being promoted.

6.2 Expected Cost

Table A.4[1] shows the number of times GPVE was able to find a fix (*count*) and the average generation it did so at (*avg*). From this information, we can calculate the expected number of generations needed, taking into account the unsuccessful runs:

$$\sum_{i=0}^{\infty}(1-p)^i \cdot p \cdot (avg + i \cdot penalty) \tag{6.1}$$

Where $p = count/30$ is the *probability* of success considering a total of 30 runs, and i is the number of times we fail before finding a fix. In these unsuccessful runs, we spend the maximum number of generations before declaring that a fix was not found, 50 for the experiments considered, which we name *penalty*.

So assuming, for instance, that a certain vulnerability was fixed 10 times, on average at generation 15, then we have $p = \frac{1}{3}$ probability of fixing it in 15 generations, plus the remaining $(1-p) = \frac{2}{3}$ of failing at least once. That is, we have a $p = (\frac{2}{3})^1 \cdot \frac{1}{3}$ of fixing it in $15 + 1 \cdot 50 = 65$ generations, after failing once ($i = 1$); a $p = (\frac{2}{3})^2 \cdot \frac{1}{3}$ of fixing it in $15 + 2 \cdot 50 = 115$ generations after failing twice ($i = 2$), and so on.

It is important to mention that the actual way these values were calculated for the table was through the formula: $\frac{avg \cdot count\ +\ 50*(30-count)}{count}$. Although this was reverse engineered

[1]Table B.4 for `trie`'s vulnerabilities

based on the intuitive expected values, Eq.6.1 was also implemented in `Python` as a sanity check.

This metric is important because it also gives us a way to estimate the expected ***time until fix***, when combined with the execution times benchmarked in Table 4.1. Since each generation takes[2] $\frac{1}{50}$ of that time for population sizes of 100, and $\frac{1}{5}$ for population sizes of 1000. For example, a baseline correction of useafterfree for population size of 100 takes an expected 46.81 generations, each taking $\frac{129.273}{50}$ seconds running on a quad core computer for an expected time until fix of approximately 121 seconds.

6.3 Fixes

We analyse some of the fixes proposed by GPVE, looking to highlight its capabilities and drawbacks. Here, we benefit from having the two straightforward variation operators, to reason about the fixes and infer the changes they underwent.

Fixes are presented as the file difference between their phenotype in green, and the original program in red. These were converted from CIL's representation, hence why they do not exactly match with Appendix A.2 and B.2's source code.

6.3.1 `skiplist`'s Double Free

This vulnerability included a loop unnecessarily freeing all of the current node's successors, eventually reaching nodes that had already been freed. Since `while(1)` and `if` are both expansions of the `stmt`(statement) nonterminal, they can be swapped between trees by selecting their parent statements as crossover point. We also observe the preference for *minimal fixes*, with GPVE pruning the explicit casts added by CIL.

Listing 6.1: Fix for `skiplist`'s doublefree.c

```
}void Free(Node *node){
      int i;
      {
-          i=0;
-          while(1){
-               if(i < 5){
-
-               }else{
-                    break;
-
-               }if((unsigned long )(*(node->next + i)) != (unsigned long )(void *)0){
-                    Free((*(node->next + i)));
+          if((*(node->next + i)) != 0){
+               Free((*(node->next + 0)));

-               }else{
-
-               }i=i + 1;
+          }else{

           }free((void *)node->next);
           free((void *)node);
```

[2]Approximately, since the number of expected generations could amount to more than one run, where no elitism is applied for their first generation

Before using tree size in acceptance criteria, it would actually *come up* with some inventive ways to skip the loop - here incrementing its counter to effectively run only one iteration, since 16 is greater than the `MAX_HEIGHT` macro's value of 5. It would similarly fix it by substituting the increment instruction with a break statement.

Listing 6.2: Fix #2 for `skiplist`'s doublefree.c

```
-              }i=i + 1;
+              }i=i + 16;

          }free((void *)node->next);
          free((void *)node);
```

6.3.2 `skiplist`'s Heap-based Buffer Overflow

Without letting it change the incorrect index initialization, GPVE actually *cheats* by traversing only through the immediate successor nodes (`node[0]`, i.e., never needing to use `i`). This has the important consequence of simplifying the algorithm to the point of affecting its complexity - search is now $O(n)$ rather than $O(\log n)$. Certainly in future work we will have to ensure the program's performance is preserved, through large test cases that would trigger a timeout otherwise.

Listing 6.3: Fix for `skiplist`'s heapbufferoverflow.c

```
      {
          while(1){
-              if(i >= 0){
-
-              }else{
-                  break;
-
-              }while(1){
-                  if((unsigned long )(*(node->next + i)) != (unsigned long )(void *)0){
-                      tmp=strcmp((char *)cartao,(char *)(*(node->next + i))->cartao);
-                      if(tmp > 0){
+              if((*(node->next + 0))){
+                  tmp=strcmp(cartao,(*(node->next + 0))->cartao);
+                  if(tmp > 0){

-                      }else{
-                          break;
-
-                      }
                  }else{
                      break;

-                  }node=(*(node->next + i));
+                  }

+              }else{
+                  break;

-              }i=i - 1;
+              }node=(*(node->next + 0));
```

6.3.3 trie's Null Pointer Dereference

This vulnerability changes an if condition from checking if a pointer is NULL to if its non NULL. The fix ends up being very simple: since an if expansion consists of the following *flattened* expansion: "if", "(", exp, ")", "{", stmtlist, "}", "else", "{", stmtlist, "}"; the branches can be swapped by the crossover selecting each of the branches of the parents' trees.

Listing 6.4: Fix for trie's nullpointer.c

```
         if(node->next[index___0]){
-            return 0;
-
-        }else{
             if(pos == len - 1){
                 return node->next[index___0]->terminal;

             }else{
                 tmp=find(node->next[index___0],in,len,pos + 1);
                 return tmp;

             }
+        }else{
+            return 0;
+
         }
```

6.3.4 trie's Buffer Overflow

The count function checks the number of words with the given prefix. The vulnerability consists simply of incorrectly checking that we are at its last letter (leading to an over-read).

This incorrectly accepted fix shows the pitfalls of the small test suites used, and of the branch coverage criteria (reported at 100%). Although count was exercised by several test cases, expecting the return value to be zero, one or more, all of the prefixes started with an 'a', ie. the first letter / index 0. Further, it did not use roots that covered only a portion of the words, making it so that an obviously incorrect program was able to pass all test cases. Test suite generation based on data flow analysis would have certainly detected such inadequacies.

Another takeaway is how it reinforces fix size's inadequecy measure for understandability. Since it significantly alters the program's control flow, it ends up overfitting on the test suite. Meanwhile had we looked to minimize the fix's editing distance to the original program, the expected fix of reverting the vulnerability would have certainly prevailed.

Listing 6.5: Fix for trie's overflow.c

```
 }int count(Node *node,char *in,int len,int pos){
     int index___0;
     int tmp;
     {
-        index___0=(int )(*(in + pos)) - 97;
-        if((unsigned long )node->next[index___0] == (unsigned long )(void *)0){
+        if(node->next[0] == 0){
             return 0;
```

```
        }else{
-           if(pos == len){
-               return node->next[index___0]->count;
-
-           }else{
-               tmp=count(node->next[index___0],in,len,pos + 1);
-               return tmp;
+           return node->next[0]->count;

        }
    }
```

6.3.5 `trie`'s Use After Free

The following incorrectly proposed fix **compromises** results observed for the functions whose sole purpose is freeing memory (Free and rfree).

Detection of memory leaks with ASan (through LeakSanitizer) is apparently still experimental, meaning its reports are inconsistent, as shown in Fig.6.6. As such, *fixes* like the one below are free to fil these functions with bogus statements - notably a `scanf` called with pointer, since we do not support typing for variadic functions, which surprinsingly compiles compilable (albeit with a warning). It does show though how removing/replacing multiple statements is straightforward, if the variation operators select the `stmtlist` containing them.

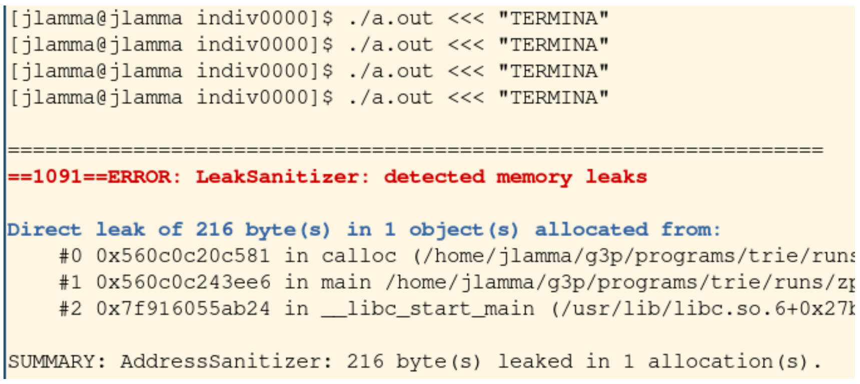
ASan's false negative reports imply that if a candidate fix is *lucky* enough to pass through the test suite without triggering the memory leak detection, then they are accepted and never again evaluated since they are preserved through elitism. And as these fixes remove most of the function's statements, they have minimal size and are thus proposed over potentially correct fixes.

This incident reinforces the need for automated test generated, (immediately) before fixes are proposed as they would run on thousands rather than tens of test cases, greatly reducing the changes that an incorrect fix such as this one would ever slip by. It also suggests the use of additional dynamic detection tools for memory leaks, eg. Valgrind.

Listing 6.6: incorrectly accepted fix for `trie`'s useafterfree.c

```
 }void rfree(Node *node){
    int i;
    {
-       i=0;
-       while(1){
-           if(i < 26){
-
-           }else{
-               break;
-
-           }if((unsigned long )node->next[i] != (unsigned long )(void *)0){
-               rfree(node->next[i]);
-               free((void *)node);
-
-           }else{
-
-           }i=i + 1;
-
−           }return ;
+           scanf(node);
+           return ;

      }
```



```
[jlamma@jlamma indiv0000]$ ./a.out <<< "TERMINA"
[jlamma@jlamma indiv0000]$ ./a.out <<< "TERMINA"
[jlamma@jlamma indiv0000]$ ./a.out <<< "TERMINA"
[jlamma@jlamma indiv0000]$ ./a.out <<< "TERMINA"

=================================================================
==1091==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 216 byte(s) in 1 object(s) allocated from:
    #0 0x560c0c20c581 in calloc (/home/jlamma/g3p/programs/trie/runs
    #1 0x560c0c243ee6 in main /home/jlamma/g3p/programs/trie/runs/zp
    #2 0x7f916055ab24 in __libc_start_main (/usr/lib/libc.so.6+0x27b

SUMMARY: AddressSanitizer: 216 byte(s) leaked in 1 allocation(s).
```

Figure 6.6: LeakSanitizer inadequacy for correction of memory leaks

Chapter 7

Conclusion

Automated program repair is a challenging endeavour. The EA-based *generate-and-validate* approaches result in an enormous search space of candidate fixes, requiring the implementation of complex constraints and heuristics to make the problem tractable. Still, this effort should pale in comparison to the time it saves for developers, lifting them from mundane correction tasks [11], and companies from the adverse impacts arising from vulnerability exploitation.

In this work we specifically targetted the repair of vulnerabilities, aiming to benefit from their simpler corrections encodable through fix patterns, in the proposal of a framework based on Genetic Programming. Revealing vulnerabilities through instrumentation allows us to focus the search onto small intervals of source code lines, further reducing the search space through the generation of syntactically and (mostly) semantically correct candidate fixes for C programs.

A set of memory access vulnerabilities were then injected into implementations of the skiplist and trie data structures. We explored GPVE's ability to generate acceptable fixes, particularly with regards to their expected cost in number of generations. Analysing the impact of the vulnerabilities' localization showed how a larger interval can actually be beneficial, due to the predominant application of subtree recombination accessing surrounding code. Promising results were also observed when assigning weights to each test case according to their perceived difficulty. Inspection of proposed fixes revealed how GPVE is able to generate complex source code modifications, although the preference for smaller genotypes encouraged simplifications resulting in changes to the underlying algorithms, that then overfitted on the reduced test suites.

Although limitations in the memory leak detection compromised the results for vulnerabilities located in the Free and rfree functions, the remaining nine vulnerabilities' best performing experiments give us an overall success rate of 87.9% in generating fixes that were evolved to pass all tests in the suite. These are extremely uplifting results, reinforcing our trust in GPVE's potential to contribute to the practicality of automated program repair tools, thus far not receiving much attention beyond academia.

Future Work

There is still a long way to go until GPVE achieves what we set out to, with immediate concerns including the support for large scale programs and automated generation of high coverage test suites.

We now leave off with some suggestions for future work:

- Identify metrics for preventing bloat, while preserving the program's non-functional properties, and familiarity in general. Tree edit distance [74] to the original program's genotype would have made more sense in hindsight, but this should be should be confirmed by surveying developers on a set of proposed fixes according to different metrics.
- Deal with imprecise vulnerability localization through dynamic code extraction, where the intervals of source code lines are encoded in the genotype and dynamically increased if the population stagnates. Further, where the vulnerability is reported may be different from where its root cause lies, and so we may end up just *delaying* it being triggered. Vulnerability reports should be monitored to detect new intervals that need to be operated on.
- Deal with programs for which not all correctness can be inferred from its `stdout`, namely by providing test cases aimed not only at exercising new paths, but also at confirming an algorithm's complexity is preserved - there should not be noticeable differences in execution time to the original program.
- Gain better insights into the generation of the proposed fixes, by tracking and replicating the changes each candidate fix underwent - could be encoded by their parent(s) and the variation operation applied - nodes' `dfs` for crossover and grammar expansions for mutation. This analysis would also open up the possibility of biasing the probability of of nonterminals when selecting mutation/crossover points, and of grammar productions when generating random subtrees, according to the type of vulnerability being fixed.

References

[1] Len Erlikh. Leveraging legacy system dollars for e-business. *IT professional*, 2(3):17–23, 2000.

[2] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.

[3] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[4] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.

[5] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[6] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, Xinying Wang, Anh Tuan Nguyen, and Tien N Nguyen. Detecting recurring and similar software vulnerabilities. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 227–230. IEEE, 2010.

[7] Raul Barbosa, Frederico Cerveira, Luis Goncalo, and Henrique Madeira. The most frequent programming mistakes that cause software vulnerabilities. *arXiv preprint arXiv:1912.01948*, 2019.

[8] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 22(3):415–432, 2017.

[9] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software quality journal*, 21(3):421–443, 2013.

[10] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[11] William B Langdon and Gabriela Ochoa. Genetic improvement: A key challenge for evolutionary computation. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 3068–3075. IEEE, 2016.

[12] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. How to design a program repair bot? insights from the repairnator project. In *2018 IEEE/ACM 40th*

International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pages 95–104. IEEE, 2018.

[13] Thomas Weise. Global optimization algorithms-theory and application. *Self-Published Thomas Weise*, 2009.

[14] Marco Vieira and Nuno Antunes. Introduction to software security concepts. In *Innovative Technologies for Dependable OTS-Based Critical Systems*, pages 29–38. Springer, 2013.

[15] G McGraw. Software security: Building security in, 2006.

[16] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[17] The Chromium Projects. Memory safety. `https://www.chromium.org/Home/chromium-security/memory-safety`, 2014. Accessed: 2021-01-19.

[18] David Hovemeyer and William Pugh. Finding bugs is easy. *Acm sigplan notices*, 39(12):92–106, 2004.

[19] MITRE. 2020 cwe top 25 most dangerous software weaknesses. *Common Weakness Enumeration*, 2020. Accessed: 2021-01-19.

[20] Paul E Black. Sard: Thousands of reference programs for software assurance. *J. Cyber Secur. Inf. Syst. Tools Test. Tech. Assur. Softw. Dod Softw. Assur. Community Pract*, 2(5), 2017.

[21] Kendra Kratkiewicz and Richard Lippmann. A taxonomy of buffer overflows for evaluating static and dynamic software testing tools. In *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, volume 500, pages 44–51, 2006.

[22] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[23] Charles A Hoare, Ole-Johan Dahl, and Edsger W Dijkstra. *Structured programming.* Acad. Press, 1990.

[24] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[25] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[26] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[27] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[28] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[29] Patrice Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 63(2):70–76, 2020.

[30] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[31] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[32] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[33] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[34] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 309–318, 2012.

[36] Anders Møller and Michael I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[37] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[38] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004.

[39] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing.* Springer, 2015.

[40] Sean Luke. *Essentials of Metaheuristics.* Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[41] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[42] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[43] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006.

[44] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[45] Peter A Whigham et al. Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, volume 16, pages 33–41, 1995.

[46] Robert I Mckay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.

[47] David J Montana. Strongly typed genetic programming. *Evolutionary computation*, 3(2):199–230, 1995.

[48] Mark Harman, William B Langdon, Yue Jia, David R White, Andrea Arcuri, and John A Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–14. IEEE, 2012.

[49] Nuno Lourenço, Filipe Assunção, Francisco B Pereira, Ernesto Costa, and Penousal Machado. Structured grammatical evolution: a dynamic approach. In *Handbook of Grammatical Evolution*, pages 137–161. Springer, 2018.

[50] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[51] Nuno Lourenço, Francisco B Pereira, and Ernesto Costa. Unveiling the properties of structured grammatical evolution. *Genetic Programming and Evolvable Machines*, 17(3):251–289, 2016.

[52] Michael O'Neill and Conor Ryan. Evolving multi-line compilable c programs. In *European Conference on Genetic Programming*, pages 83–92. Springer, 1999.

[53] Robin Harper and Alan Blair. A structure preserving crossover in grammatical evolution. In *2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2537–2544. IEEE, 2005.

[54] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[55] William B Langdon and Mark Harman. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, 2014.

[56] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[57] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[58] Saemundur O Haraldsson, John R Woodward, Alexander EI Brownlee, and Kristin Siggeirsdottir. Fixing bugs in your sleep: how genetic improvement became an overnight success. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1513–1520, 2017.

[59] Andrea Arcuri. Evolutionary repair of faulty software. *Applied soft computing*, 11(4):3494–3514, 2011.

[60] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 162–168. IEEE, 2008.

[61] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[62] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31, 2017.

[63] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 269–278. IEEE, 2019.

[64] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[65] Muhammad Sheraz Anjum and Conor Ryan. Seeding grammars in grammatical evolution to improve search based software testing. In *European Conference on Genetic Programming (Part of EvoStar)*, pages 18–34. Springer, 2020.

[66] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 256–267, 2017.

[67] Manuel López-Ibáñez, Juergen Branke, and Luís Paquete. Reproducibility in evolutionary computation. *arXiv preprint arXiv:2102.03380*, 2021.

[68] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. Retrofitting parallelism onto ocaml. *arXiv preprint arXiv:2004.11663*, 2020.

[69] KC Sivaramakrishnan. Effective concurrency with algebraic effects. `https://kcsrk.info/ocaml/multicore/2015/05/20/effects-multicore/`, 2015. Accessed: 2021-10-25.

[70] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[71] Simon Colton. Computational discovery in pure mathematics. In *Computational discovery of scientific knowledge*, pages 175–201. Springer, 2007.

[72] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[73] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[74] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.

Appendices

This page is intentionally left blank.

Appendix A

Program - Skiplist

A.1 Vulnerabilities

Vulnerability	Weakness	AddressSanitizer
doublefree	CWE-476	Line 102: "SEGV on unknown address"
freenondynamic	CWE-590	Line 95: "attempting to free on address which was not malloc()-ed"
heapbufferoverflow	CWE-122	Line 70: "heap-buffer-overflow on address 0x..."
memoryleak	CWE-401	"LeakSanitizer: detected memory leaks ... allocated from ... NewNode"
nullpointer	CWE-476	Line 53: "SEGV ... address points to the zero page"
nullpointerr	CWE-476	Line 55: "SEGV ... address points to the zero page"
stackbufferoverflow	CWE-121	Line 86: "stack-buffer-overflow on address 0x..."
useafterfree	CWE-416	Line 96: "heap-use-after-free on address ... freed by thread T0 here ..."

Table A.1: CWE and summary of ASan's report for `skiplist`'s vulnerabilities

Vulnerability	Interval	Alternate
doublefree	[101 103]	
freenondynamic	[95 97]	
heapbufferoverflow	[70 72]	[69 72]
memoryleak	[100 104]	[103 103]
nullpointer	[51 53]	
nullpointerr	[55 56]	[51 56]
stackbufferoverflow	[81 86]	[86 86]
useafterfree	[90 96]	

Table A.2: Location Intervals for `skiplist`'s vulnerabilities

Vulnerability	Nonterminal	Expansion
heapbufferoverflow	exp	varname, " - ", "1"
memoryleak	instr	"free", "(", lval, ")"
nullpointerr	exp	lval, " != ", "0"

Table A.3: Fix Patterns for `skiplist`'s vulnerabilities

Listing A.1: doublefree.c diff

```
@@ -101,2 +101,4 @@ void Free(Node* node) {
–       if(node->next[0] != NULL)
–               Free(node->next[0]);
+       for(int i=0; i<MAXHEIGHT; i++){
+               if(node->next[i] != NULL)
+                       Free(node->next[i]);
+       }
```

Listing A.2: freenondynamic.c diff

```
@@ -94,0 +95 @@ void Remove(Node* node, char* cartao)
+               free(updateNext);
```

Listing A.3: heapbufferoverflow.c diff

```
@@ -69 +69 @@ void Saldo(Node* node, char* cartao)
–       for(i=MAXHEIGHT-1; i>=0; i–){
+       for(i=MAXHEIGHT; i>=0; i–){
```

Listing A.4: memoryleak.c diff

```
@@ -104 +103,0 @@ void Free(Node* node) {
–       free(node);
```

Listing A.5: nullpointer.c diff

```
@@ -51,2 +51,3 @@ void Update(Node* node, char* cartao,int valor)
–               while(node->next[i]!=NULL && strcmp(cartao,node->next[i]->cartao)>0)
+               do {
                        node=node->next[i];
+               }while(node->next[i]!=NULL && strcmp(cartao,node->next[i]->cartao)>0);
```

Listing A.6: nullpointerr.c diff

```
@@ -55 +55 @@ void Update(Node* node, char* cartao,int valor)
–       if(node->next[0]!=NULL && strcmp(node->next[0]->cartao,cartao)==0)
+       if(strcmp(node->next[0]->cartao,cartao)==0)
```

Listing A.7: stackbufferoverflow.c diff

```
@@ -81,6 +81,6 @@ void Remove(Node* node, char* cartao)
–       int i;
+       int i,j=MAXHEIGHT;
        Node* updateNext[MAXHEIGHT];
        for(i=MAXHEIGHT-1; i>=0; i--){
                while(node->next[i]!=NULL && strcmp(cartao,node->next[i]->cartao)>0)
                        node=node->next[i];
–               updateNext[i]=node;
+               updateNext[j]=node;
```

Listing A.8: useafterfree.c diff

```
@@ -90 +90 @@ void Remove(Node* node, char* cartao)
–               for(i=0; i<MAXHEIGHT; i++){
+               for(i=1; i<MAXHEIGHT; i++){
```

A.2 Original Program

Listing A.9: `skiplist` program

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define MAXHEIGHT 5
typedef struct SkipListNode
{
    char cartao[16];
    int saldo;
    int height;
    struct SkipListNode** next;
}Node;

void Imprime(Node* node)
{
    while(node->next[0]!=NULL){
        node=node->next[0];
        printf("%s SALDO %d\n", node->cartao, node->saldo);
    }
}

Node* NewNode(char* cartao, int saldo)
{
    Node* node = (Node *)malloc(sizeof(Node));
    strcpy(node->cartao,cartao);
    node->saldo=saldo;
    node->height=height();
    node->next=malloc(sizeof(Node*)*MAXHEIGHT);
    int i;
    for(i=0; i<node->height; i++)
        node->next[i]=NULL;
    return node;
}

int height()
{
    int altura=1;
    srand(time(NULL));
    while(rand()%2==0 && altura<MAXHEIGHT)
        altura++;
    // printf("%d\n",altura);
    return altura;
}

void Update(Node* node, char* cartao,int valor)
{
    int i;
    Node* updateNext[MAXHEIGHT];
    for(i=MAXHEIGHT-1; i>=0; i--){
        while(node->next[i]!=NULL && strcmp(cartao,node->next[i]->cartao)>0)
            node=node->next[i];
        updateNext[i]=node;
    }
    if(node->next[0]!=NULL && strcmp(node->next[0]->cartao,cartao)==0)
        node->next[0]->saldo+=valor;
    else{
        Node* novo = NewNode(cartao,valor);
        for(i=0; i<novo->height; i++){
            novo->next[i]=updateNext[i]->next[i];
            updateNext[i]->next[i]=novo;
        }
    }
}

void Saldo(Node* node, char* cartao)
{
    int i;
    for(i=MAXHEIGHT-1; i>=0; i--){
        while(node->next[i]!=NULL && strcmp(cartao,node->next[i]->cartao)>0 )
            node=node->next[i];
    }
    if(node->next[0]==NULL || strcmp(node->next[0]->cartao,cartao)!=0)
        printf("%s INEXISTENTE\n",cartao);
    else
        printf("%s SALDO %d\n",node->next[0]->cartao, node->next[0]->saldo);
}

void Remove(Node* node, char* cartao)
{
    int i;
    Node* updateNext[MAXHEIGHT];
    for(i=MAXHEIGHT-1; i>=0; i--){
        while(node->next[i]!=NULL && strcmp(cartao,node->next[i]->cartao)>0)
            node=node->next[i];
        updateNext[i]=node;
    }
    if(node->next[0]!=NULL && strcmp(node->next[0]->cartao,cartao)==0){
        Node* temp = node->next[0];
        for(i=0; i<MAXHEIGHT; i++){
            if(updateNext[i]->next[i]!=temp)
                break;
            updateNext[i]->next[i]=temp->next[i];
        }
        free(temp->next);
        free(temp);
    }
}

void Free(Node* node) {
    if(node->next[0] != NULL)
        Free(node->next[0]);
    free(node->next);
    free(node);
}

int main()
{
    int i;
    Node* header=malloc(sizeof(Node));
    header->next=malloc(sizeof(Node*)*MAXHEIGHT);
    header->height=MAXHEIGHT;
    for(i=0; i<MAXHEIGHT; i++)
        header->next[i]=NULL;
    char comando[10];
    char cartao[20];
    int valor;
    do {
        scanf("%s", comando);
        if (strcmp(comando, "UPDATE") == 0) {
            scanf("%s", cartao);
            scanf("%d", &valor);
            Update(header, cartao, valor);
        }
        else if (strcmp(comando, "REMOVE") == 0) {
            scanf("%s", cartao);
            Remove(header, cartao);
        }
        else if (strcmp(comando, "SALDO") == 0) {
            scanf("%s", cartao);
            Saldo(header,cartao);
        }
        else if (strcmp(comando, "IMPRIME") == 0) {
            Imprime(header);
        }
    } while (strcmp(comando, "TERMINA") != 0);
    Free(header);
    return 0;
}
```

A.3 Expected Fix Cost

Experiment	Vulnerability	Avg	Std	Max	Count (/30)	Expected
Baseline	doublefree	2.53	3.12	17	30	2.53
	freenondynamic	4.09	5.93	24	23	19.30
	heapbufferoverflow	N/A	N/A	N/A	0	N/A
	memoryleak	14.00	0.00	14	3	464.00
	nullpointer	8.75	7.96	29	8	146.25
	nullpointerr	2.00	1.41	4	3	452.00
	stackbufferoverflow	11.60	9.86	38	15	61.60
	useafterfree	3.06	2.75	12	16	46.81
Large Population	doublefree	1.00	0.00	1	30	1.00
	freenondynamic	1.00	0.00	1	30	1.00
	heapbufferoverflow	3.00	0.00	3	1	1453.00
	memoryleak	7.78	1.75	11	27	13.33
	nullpointer	9.22	7.72	37	23	24.43
	nullpointerr	4.38	4.97	16	16	48.12
	stackbufferoverflow	4.00	7.74	45	30	4.00
	useafterfree	1.43	0.56	3	30	1.43
Fix Paterns	heapbufferoverflow	N/A	N/A	N/A	0	N/A
	memoryleak	18.43	9.41	39	7	182.71
	nullpointer	7.10	1.76	10	10	107.10
Alternative Locations	heapbufferoverflow	3.55	3.23	14	22	21.73
	memoryleak	12.12	4.11	17	8	149.62
	nullpointerr	6.86	7.38	21	7	171.14
	stackbufferoverflow	1.30	0.90	4	10	101.30
Dynamic Fitness	doublefree	4.07	7.58	41	30	4.07
	freenondynamic	5.24	9.72	41	17	43.47
	heapbufferoverflow	41.50	5.50	47	2	741.50
	memoryleak	18.00	0.00	18	1	1468.00
	nullpointer	18.00	16.87	47	4	343.00
	nullpointerr	1.00	0.00	1	1	1451.00
	stackbufferoverflow	5.55	3.92	13	11	91.91
	useafterfree	3.95	4.47	19	19	32.89
Dyn. Fitness w/ Large Pop.	heapbufferoverflow	31.75	7.86	50	12	106.75
	memoryleak	9.84	8.77	46	19	38.79
	nullpointer	7.12	4.82	25	25	17.12
	nullpointerr	6.67	11.11	42	15	56.67

Table A.4: Expected cost (in generations) until fix for `skiplist`'s vulnerabilities

A.4 Results

A.4.1 Baseline

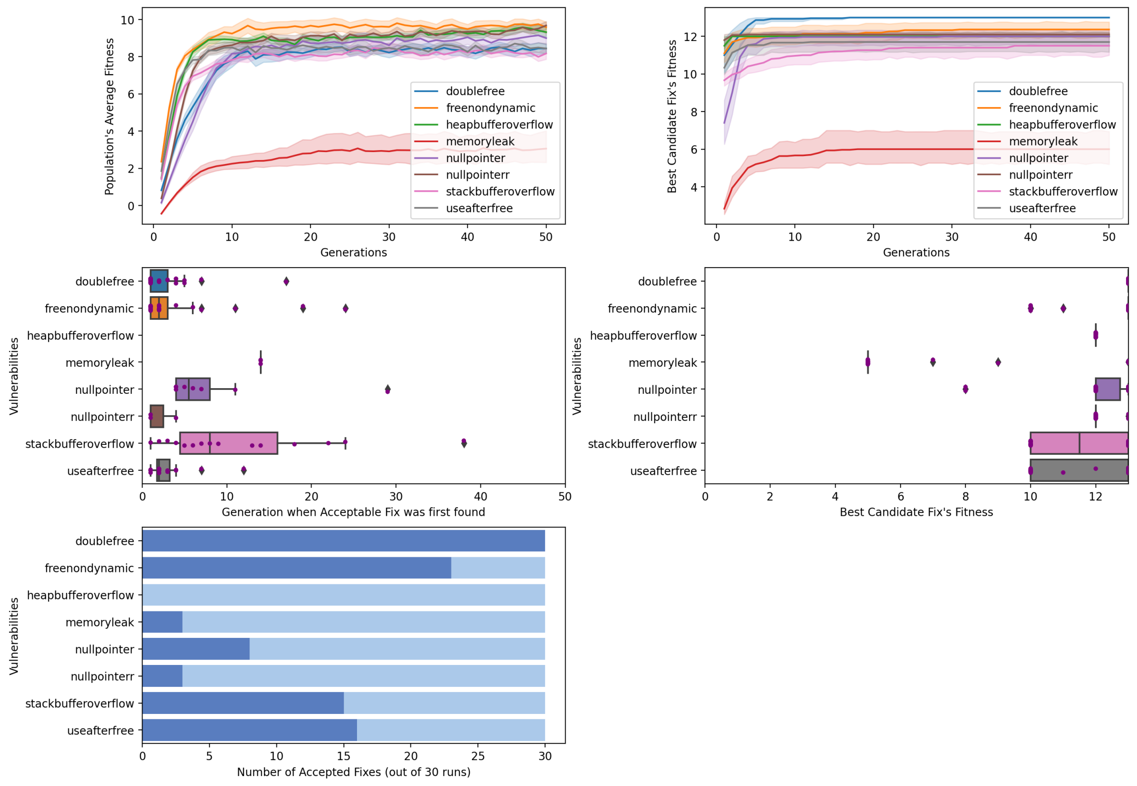


Figure A.1: skiplist pop100

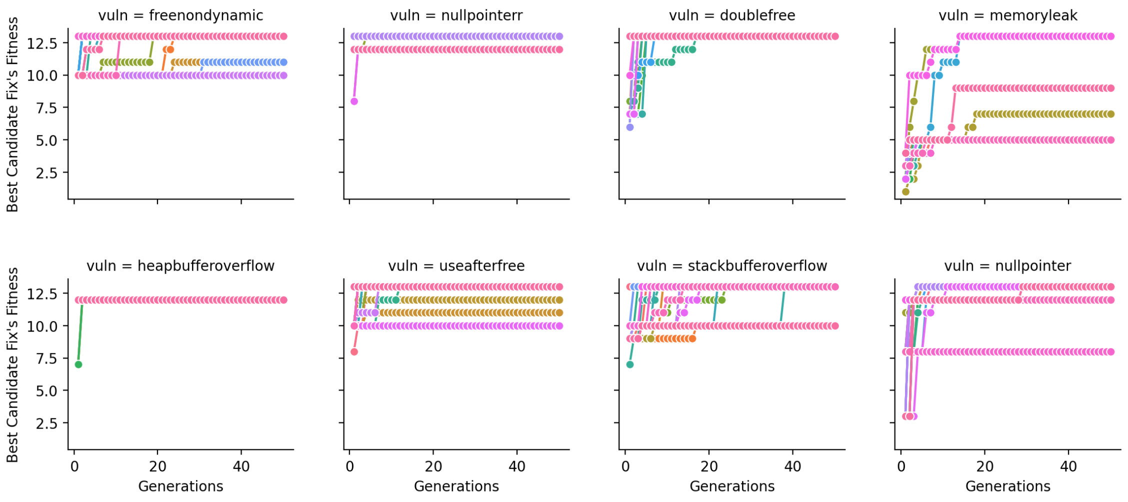


Figure A.2: skiplist pop100

A.4.2 *Large* Population

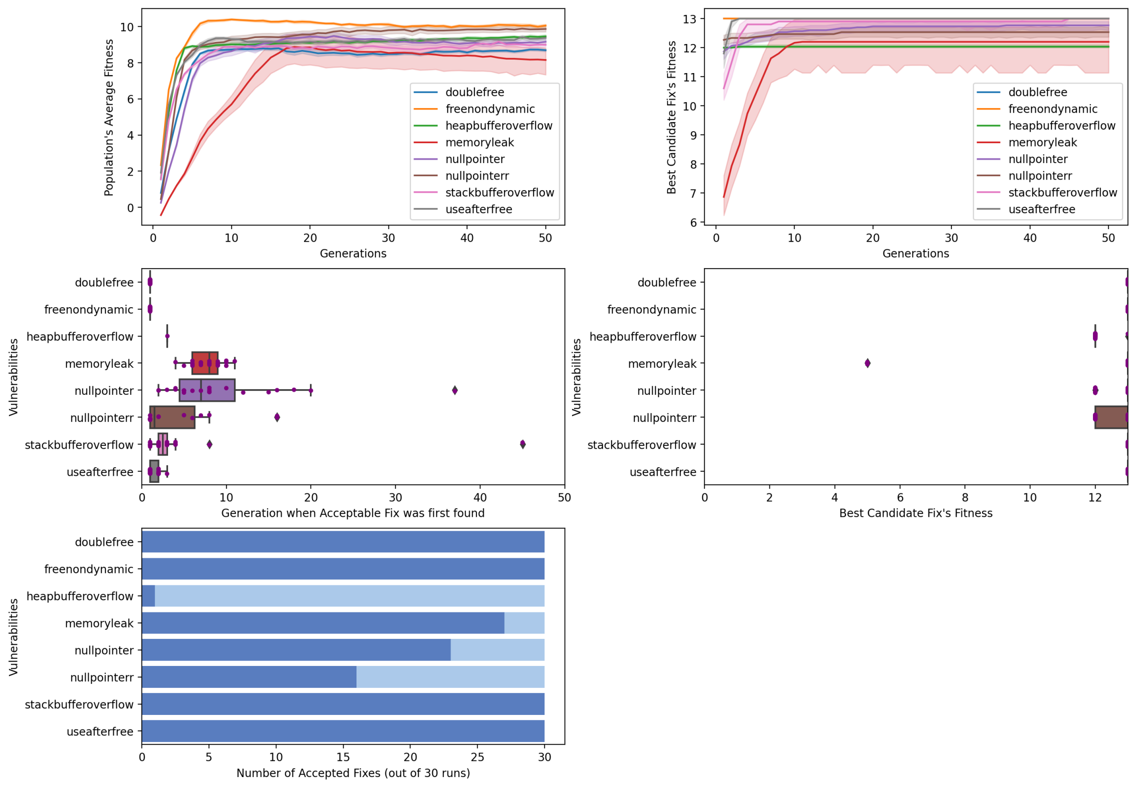


Figure A.3: skiplist pop1k

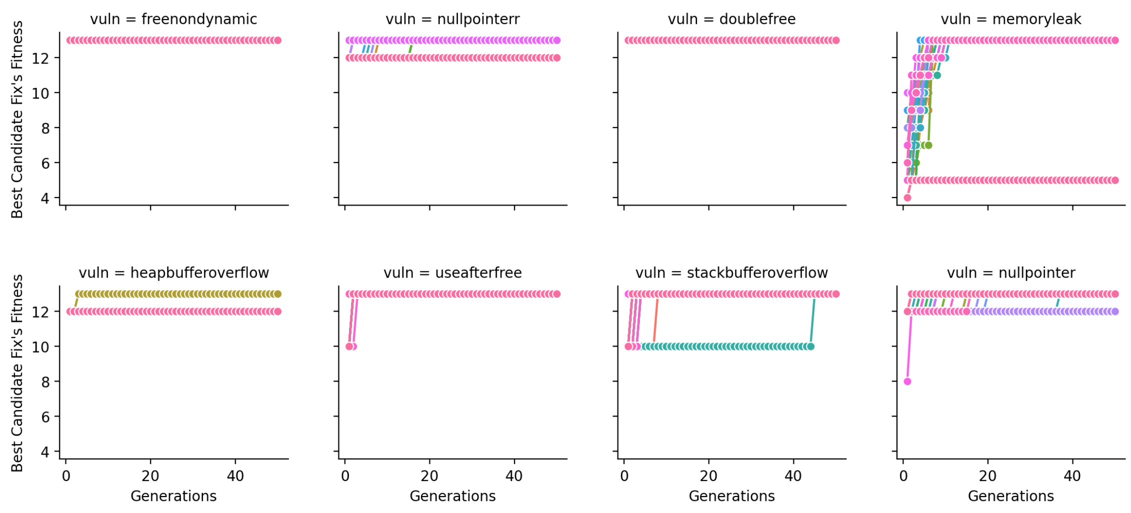


Figure A.4: skiplist pop1k

A.4.3 Fix Patterns

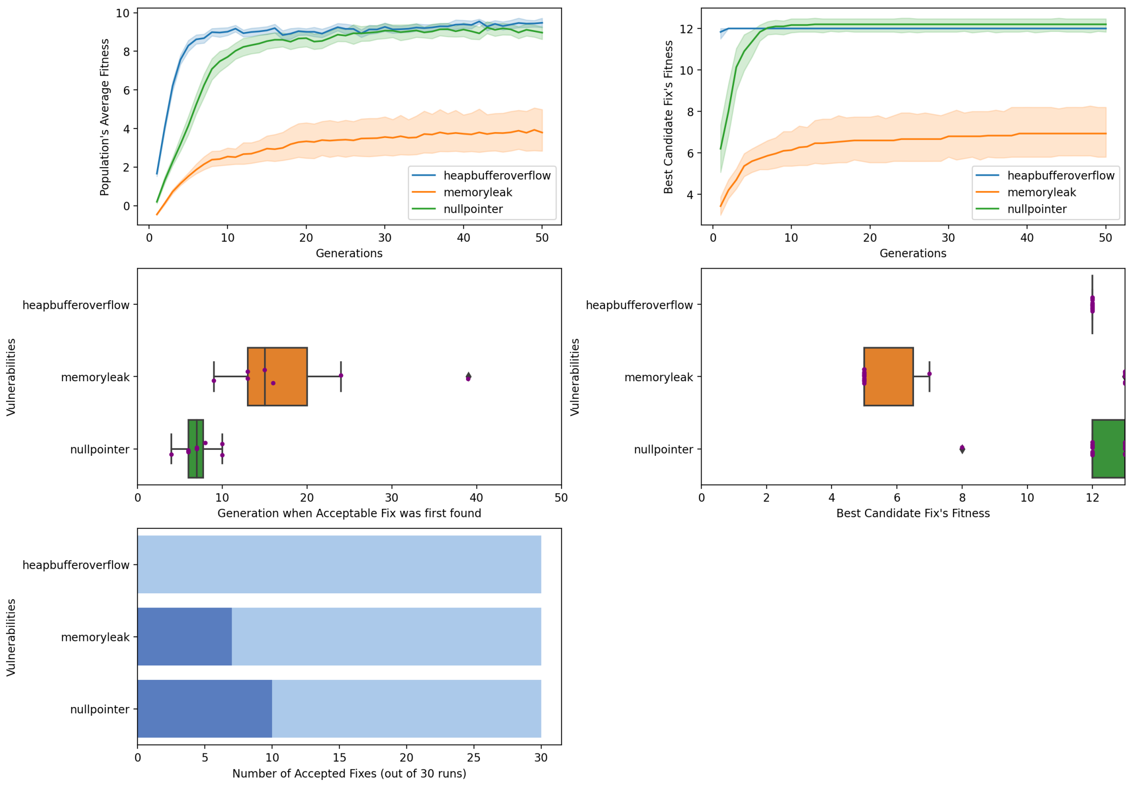


Figure A.5: skiplist pop100patterns

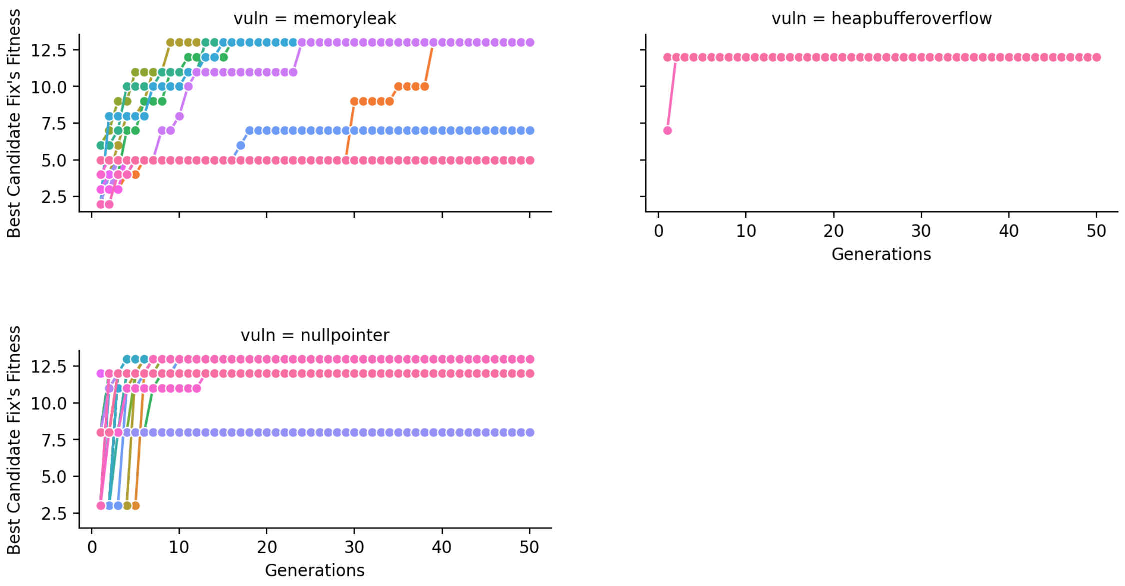


Figure A.6: skiplist pop100patterns

A.4.4 Alternative Locations

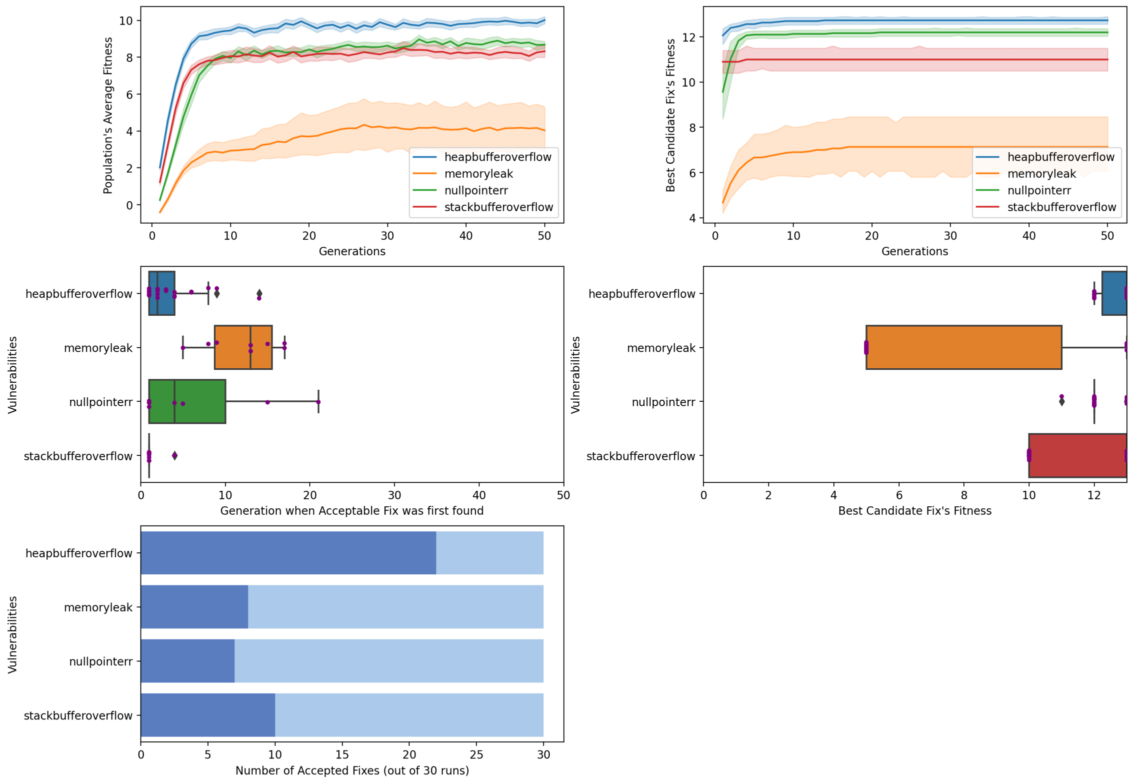


Figure A.7: skiplist pop100altlocs

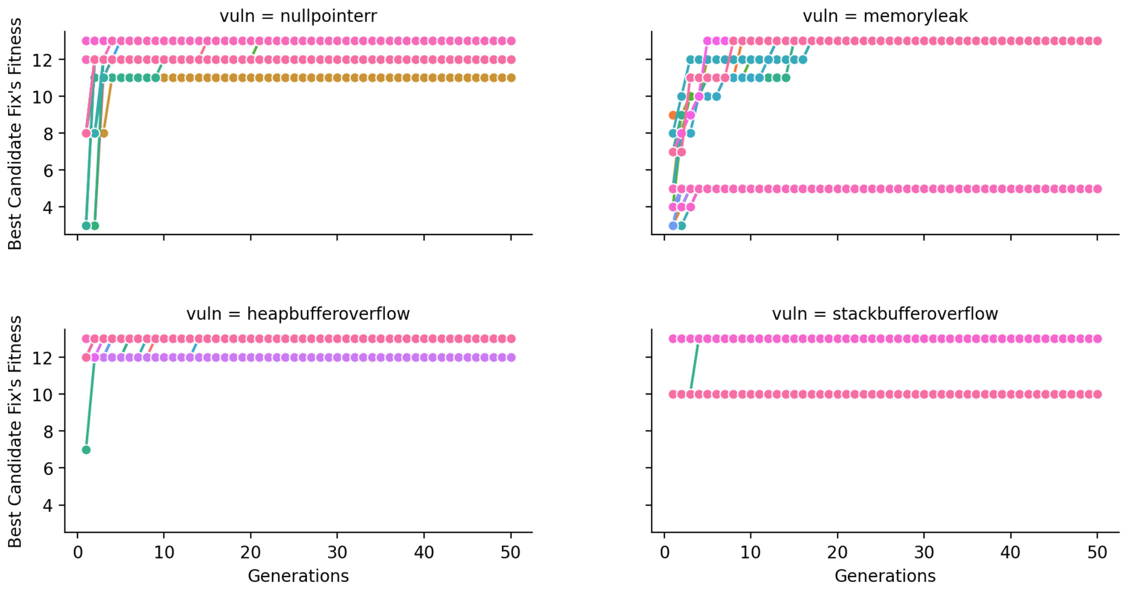


Figure A.8: skiplist pop100altlocs

A.4.5 Dynamic Fitness

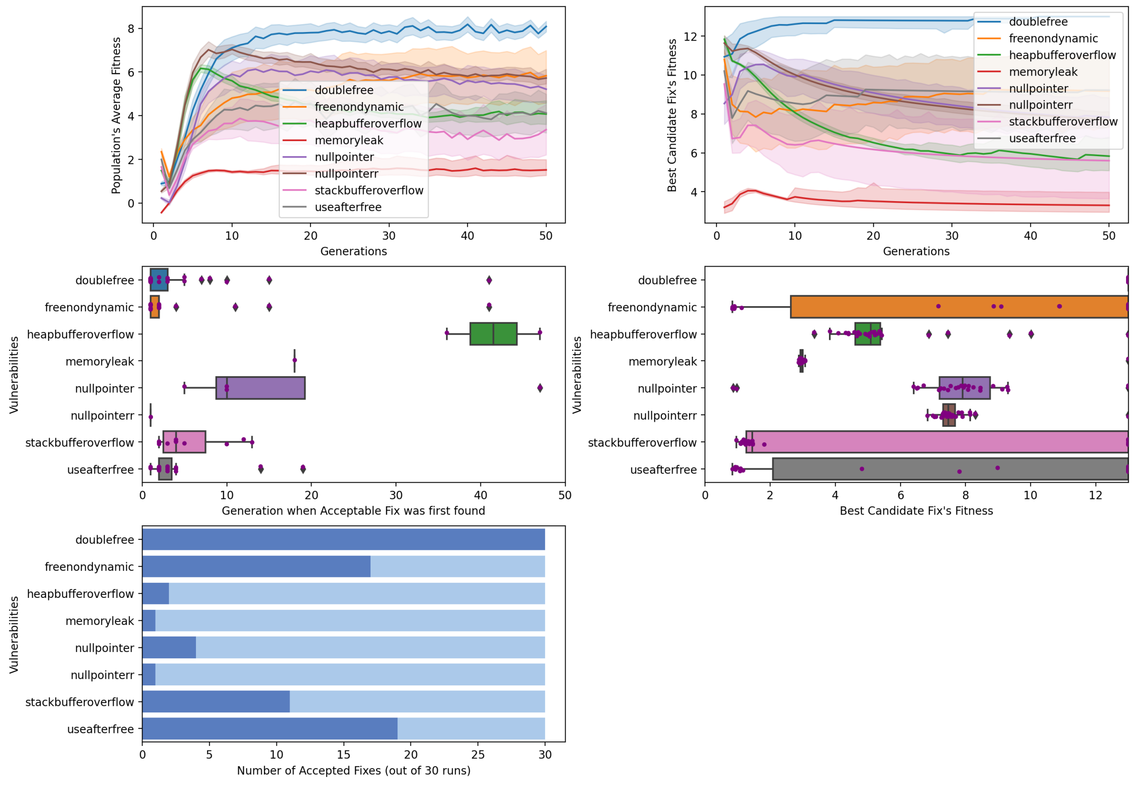


Figure A.9: skiplist pop100dynamicfitness

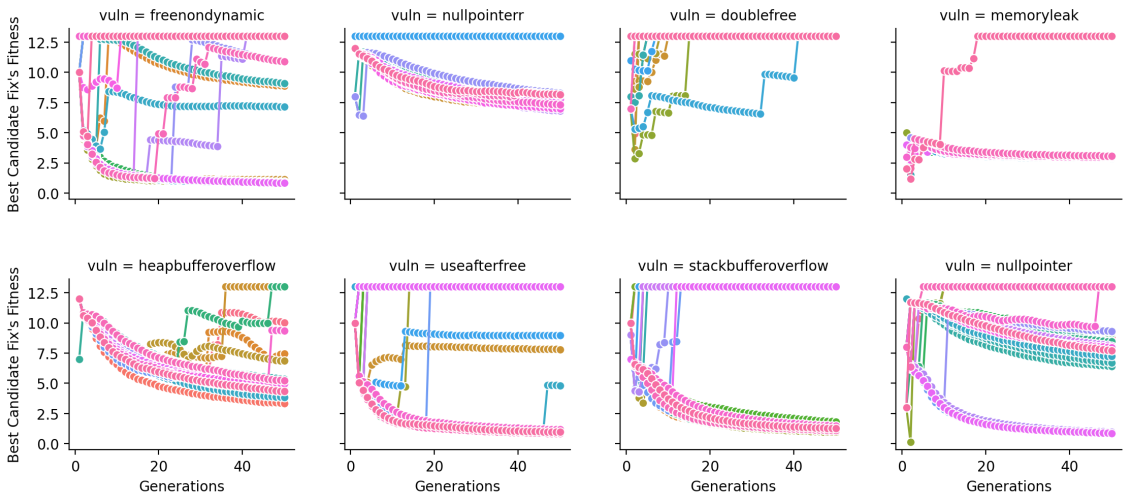


Figure A.10: skiplist pop100dynamicfitness

A.4.6 Dynamic Fitness with *Large* Population

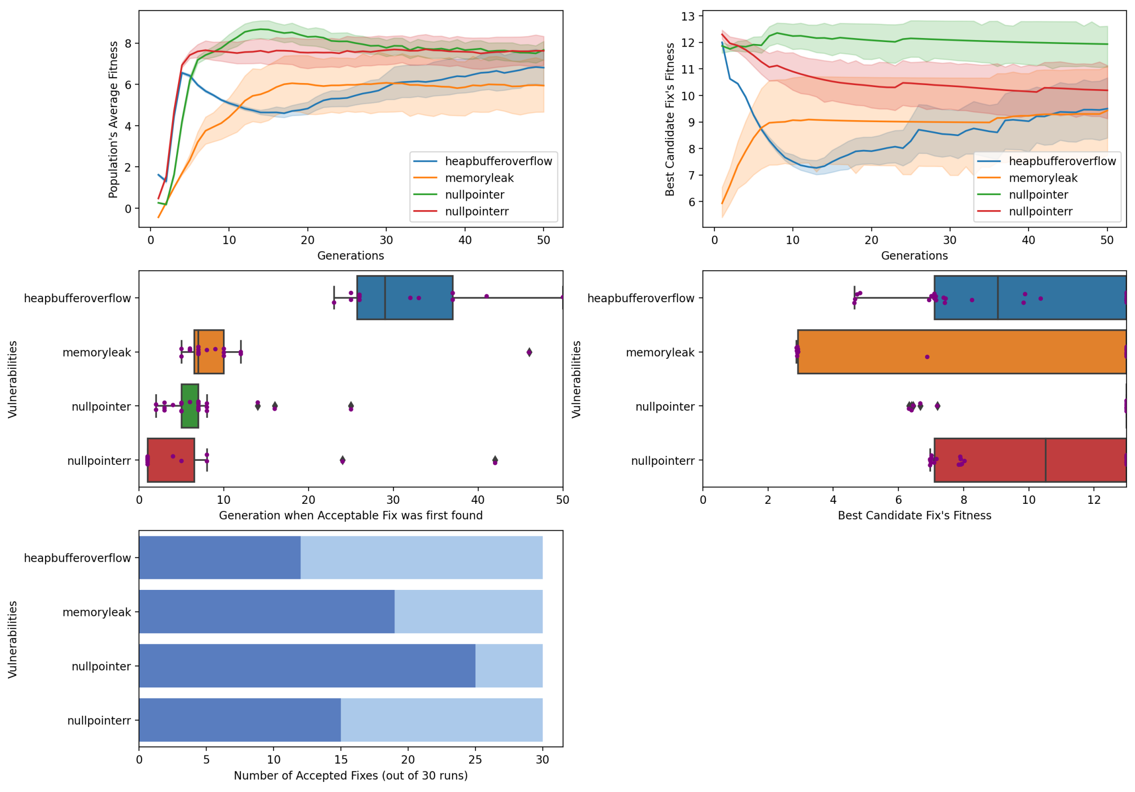


Figure A.11: skiplist pop1kdynamicfitness

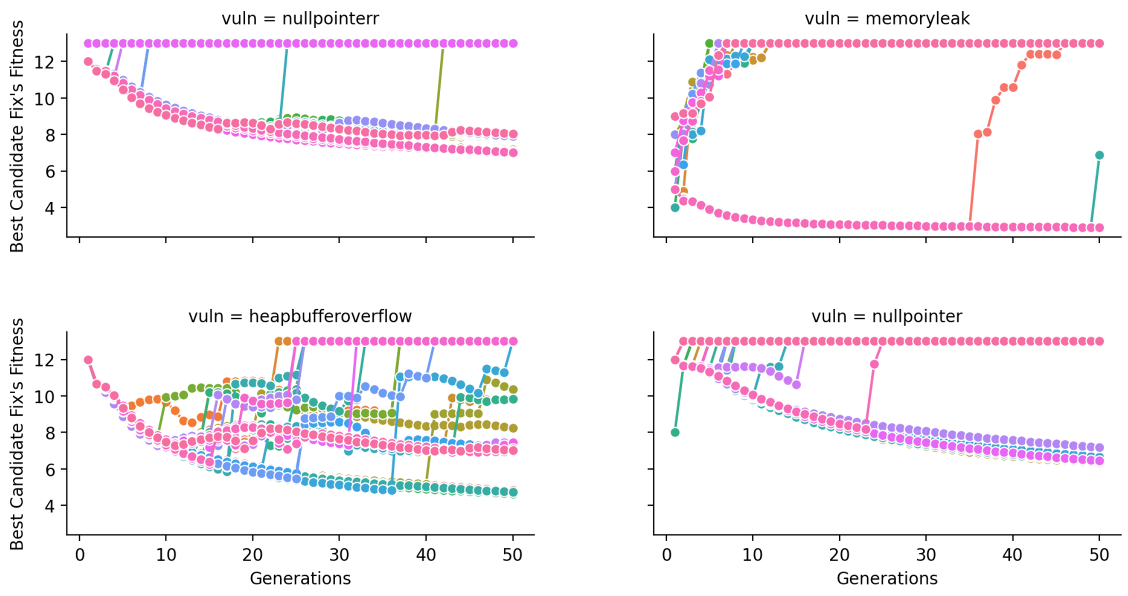


Figure A.12: skiplist pop1kdynamicfitness

Appendix B

Program - Trie

B.1 Vulnerabilities

Vulnerability	Weakness	AddressSanitizer
doublefree	CWE-415	Line 72: "attempting double-free"
nullpointer	CWE-476	Line 26: "SEGV ... address points to the zero page"
overflow	CWE-126	Line 35: "SEGV on unknown address" (called from Line 40)
useafterfree	CWE-416	Line 70: "heap-use-after-free ... freed by thread T0 here"
useafterfreee	CWE-416	Line 50: "heap-use-after-free ... freed by thread T0 here"

Table B.1: CWE and summary of ASan's report for `trie`'s vulnerabilities

Vulnerability	Interval	Alternate
doublefree	[71 72]	[66 72]
nullpointer	[26 31]	[26 27]
overflow	[34 38]	[34 40]
useafterfree	[67 72]	
useafterfreee	[49 50]	

Table B.2: Location Intervals for `trie`'s vulnerabilities

Vulnerability	Nonterminal	Expansion
nullpointer	exp	lval, " == ", "0"
overflow	exp	lval, " == ", varname, " - ", "1"
useafterfreee	instr	lval, "=", "0"

Table B.3: Fix Patterns for `trie`'s vulnerabilities

Listing B.1: doublefree.c diff

```
@@ -70,0 +71 @@ void rfree(Node* node){
+       free(node->next);
```

Listing B.2: nullpointer.c diff

```
@@ -26 +26 @@ int find(Node* node, char* in, int len, int pos){
-       if(node->next[index]==NULL)
+       if(node->next[index])
```

Listing B.3: overflow.c diff

```
@@ -37 +37 @@ int count(Node* node, char* in, int len, int pos){
-       else if(pos == len-1)
+       else if(pos == len)
```

Listing B.4: useafterfree.c diff

```
@@ -67,5 +67,6 @@ void rfree(Node* node){
        for(i=0;i<26;i++)\{
-               if(node->next[i]!=NULL)
+               if(node->next[i]!=NULL){
                        rfree(node->next[i]);
+                       free(node);
+               }
        }
-       free(node);
```

Listing B.5: useafterfreee.c diff

```
@@ -51 +50,0 @@ void delete(Node* node, char* in, int len, int pos){
-               node->next[index]=NULL;
```

B.2 Original Program

Listing B.6: `trie` program

```
// code from notes for a programming competition
// adapted so that main() would be similar to skiplist's

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct node{
    struct node* next[26];
    int terminal;
    int count;
}Node;

void insert(Node* node, char* in, int len, int pos){
    int index = in[pos]-'a';
    if(node->next[index]==NULL)
        node->next[index] = (Node*)calloc(1,sizeof(Node));
    node->next[index]->count++;
    if(pos == len-1)
        node->next[index]->terminal = 1;
    else
        insert(node->next[index], in, len, pos+1);
}
int find(Node* node, char* in, int len, int pos){
    int index = in[pos]-'a';
    if(node->next[index]==NULL)
        return 0;
    else if(pos == len-1)
        return node->next[index]->terminal;
    else
        return find(node->next[index], in, len, pos+1);
}
int count(Node* node, char* in, int len, int pos){
    int index = in[pos]-'a';
    if(node->next[index]==NULL)
        return 0;
    else if(pos == len-1)
        return node->next[index]->count;
    else
        return count(node->next[index], in, len, pos+1);
}
void delete(Node* node, char* in, int len, int pos){
    int index = in[pos]-'a';
    node->next[index]->count--;
    if(pos==len-1)
        node->next[index]->terminal=0;
    else
        delete(node->next[index], in, len, pos+1);
    if(node->next[index]->count==0){
        free(node->next[index]);
        node->next[index]=NULL;
    }
}
void print(Node* node){
    int i;
    printf("[");
    for(i=0;i<26;i++){
        if(node->next[i]!=NULL){
            printf("%c", 'a'+i);
            print(node->next[i]);
        }
    }
    printf("]");
}
void rfree(Node* node){
    int i;
    for(i=0;i<26;i++){
        if(node->next[i]!=NULL)
            rfree(node->next[i]);
    }
    free(node);
}

int main(void) {
    Node* root = (Node*)calloc(1, sizeof(Node));
    char comando[10];
    char cartao[20];
    int len, valor;
    do {
        scanf("%s", comando);
        if (strcmp(comando, "INSERT") == 0){
            scanf("%s", cartao);
            len = strlen(cartao);
            insert(root, cartao, len, 0);
        }
        else if(strcmp(comando, "FIND") == 0){
            scanf("%s", cartao);
            len = strlen(cartao);
            if(find(root, cartao, len, 0))
                printf("%s EXISTENTE\n", cartao);
            else
                printf("%s INEXISTENTE\n", cartao);
        }
        else if(strcmp(comando, "COUNT") == 0){
            scanf("%s", cartao);
            len = strlen(cartao);
            valor = count(root, cartao, len, 0);
            printf("%s %d\n", cartao, valor);
        }
        else if(strcmp(comando, "DELETE") == 0){
            scanf("%s", cartao);
            len = strlen(cartao);
            if(find(root,cartao, len, 0))
                delete(root, cartao, len, 0);
        }
        else if(strcmp(comando, "PRINT") == 0){
            print(root);
            printf("\n");
        }
    } while (strcmp(comando, "TERMINA") != 0);
    rfree(root);
    return 0;
}
```

B.3 Expected Fix Cost

Experiment	Vulnerability	Avg	Std	Max	Count (/30)	Expected
Baseline	doublefree	1.07	0.25	2	30	1.07
	nullpointer	17.60	12.92	40	10	117.60
	overflow	2.43	1.52	6	30	2.43
	useafterfree	23.78	13.20	49	9	140.44
	useafterfreee	1.00	0.00	1	30	1.00
Large Population	doublefree	1.00	0.00	1	30	1.00
	nullpointer	8.13	3.85	23	30	8.13
	overflow	1.00	0.00	1	30	1.00
	useafterfree	8.55	6.33	33	29	10.28
	useafterfreee	1.00	0.00	1	30	1.00
Fix Patterns	nullpointer	16.14	12.24	41	7	180.43
	overflow	1.67	1.01	5	30	1.67
	useafterfreee	1.00	0.00	1	30	1.00
Alternative Locations	doublefree	3.20	4.02	18	30	3.20
	nullpointer	N/A	N/A	N/A	0	N/A
	overflow	2.33	1.85	10	30	2.33
Dynamic Fitness	doublefree	1.03	0.18	2	30	1.03
	nullpointer	20.50	14.18	44	12	95.50
	overflow	1.70	0.78	3	30	1.70
	useafterfree	25.44	15.69	50	9	142.11
	useafterfreee	1.00	0.00	1	30	1.00

Table B.4: Expected cost (in generations) until fix for `trie`'s vulnerabilities

B.4 Results

B.4.1 Baseline

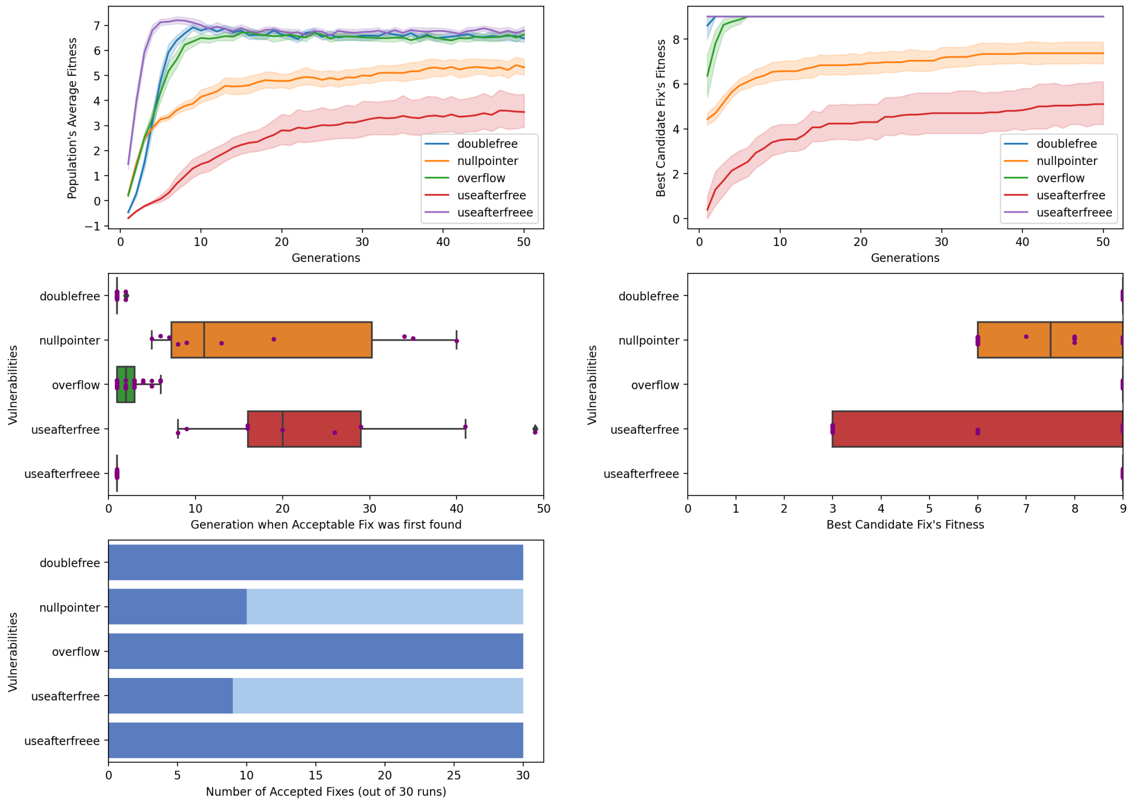


Figure B.1: trie pop100

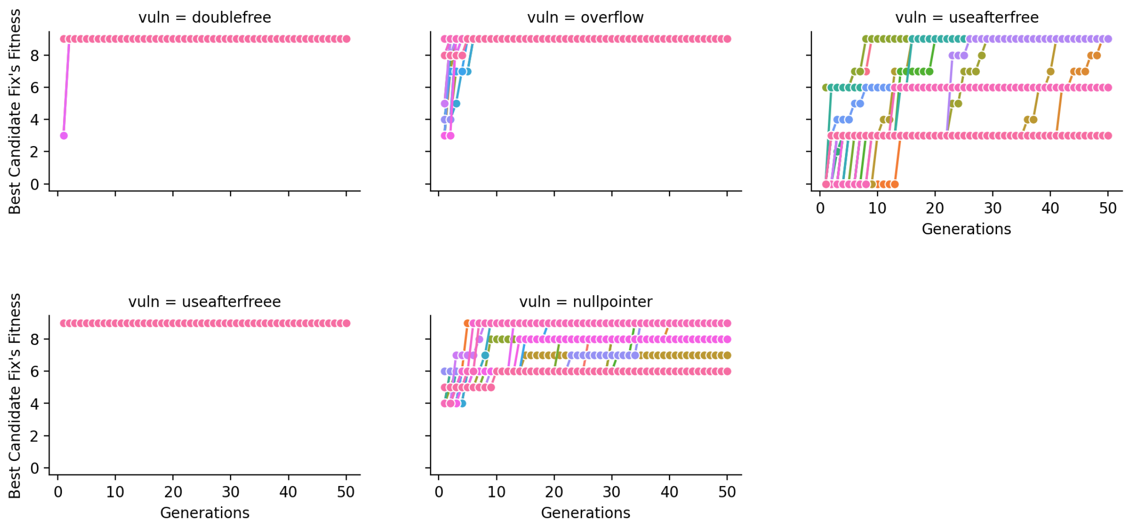


Figure B.2: trie pop100

B.4.2 *Large* Population

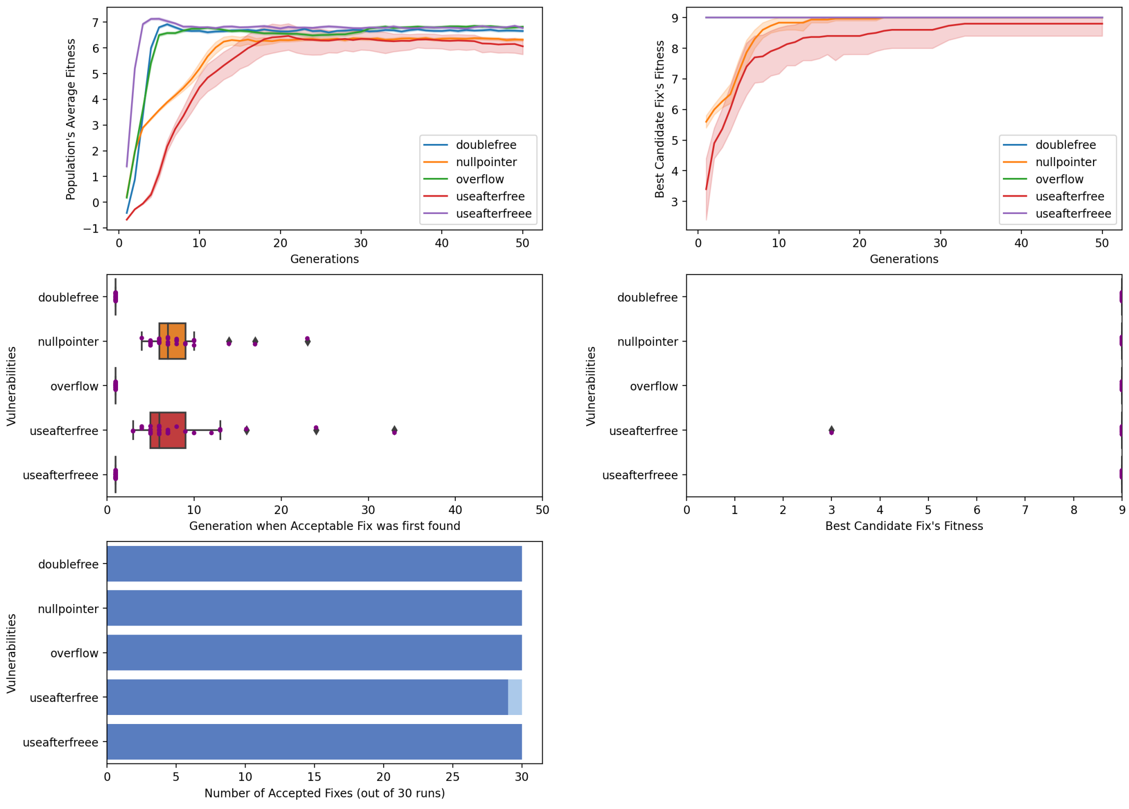


Figure B.3: trie pop1k

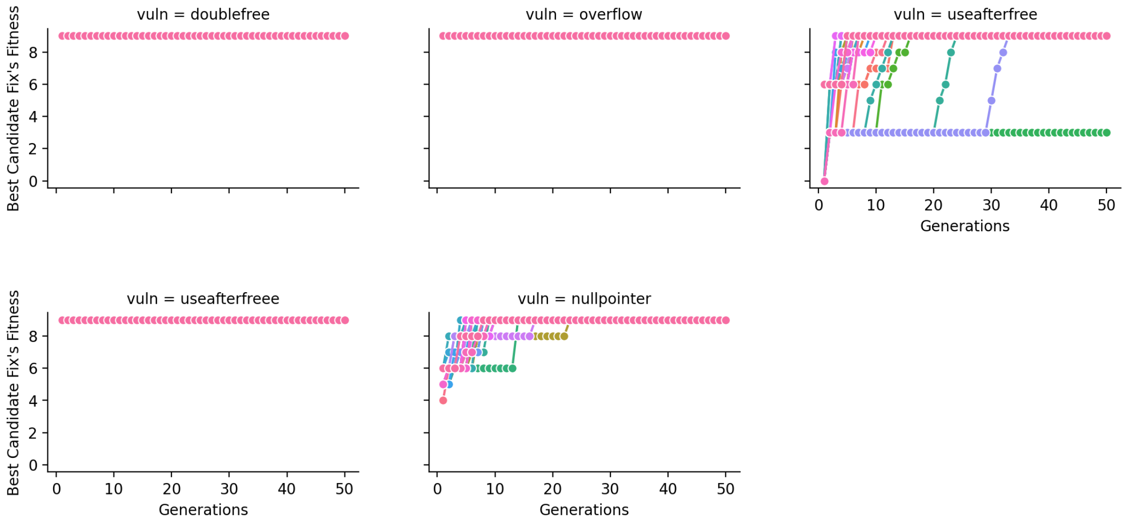


Figure B.4: trie pop1k

B.4.3 Fix Patterns

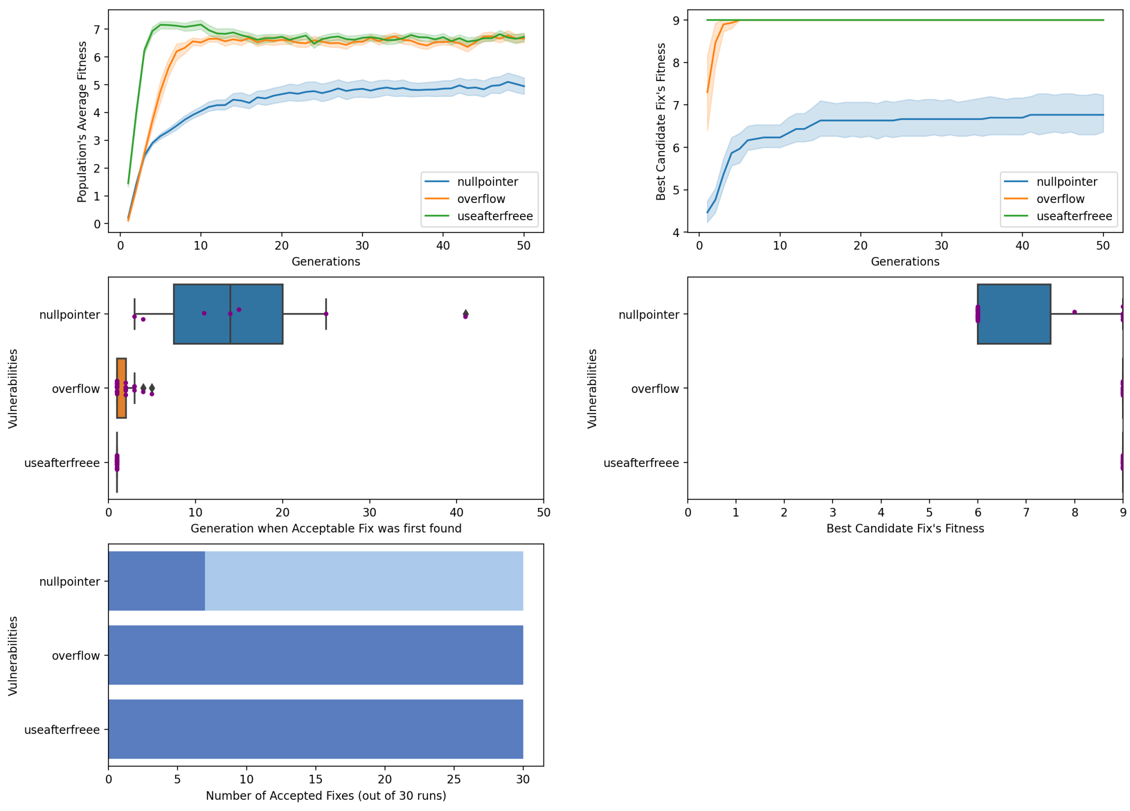


Figure B.5: trie pop100patterns

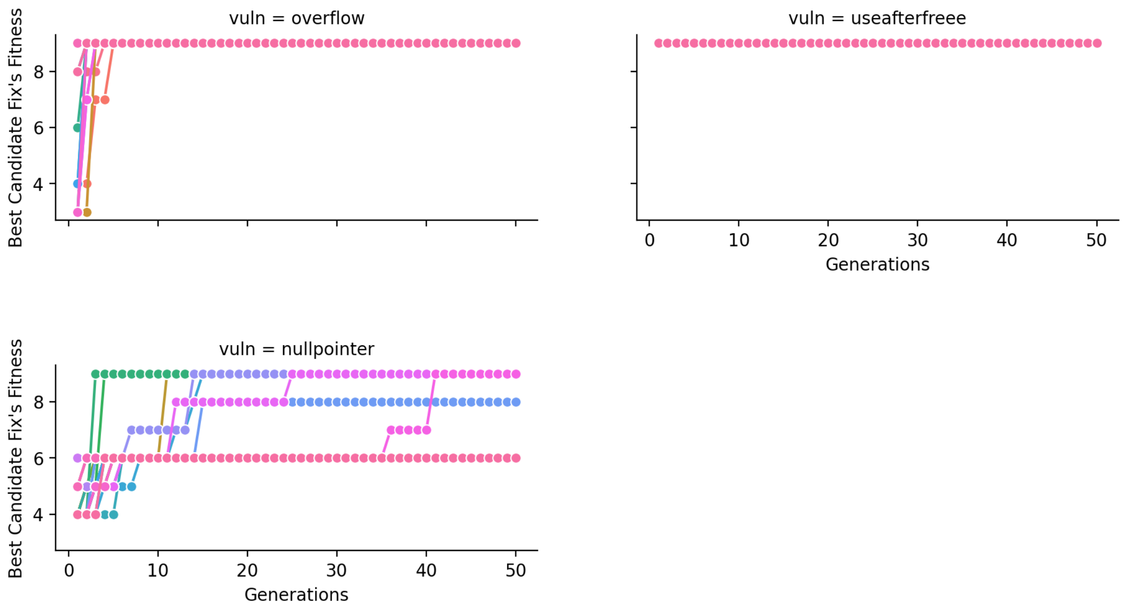


Figure B.6: trie pop100patterns

B.4.4 Alternative Locations

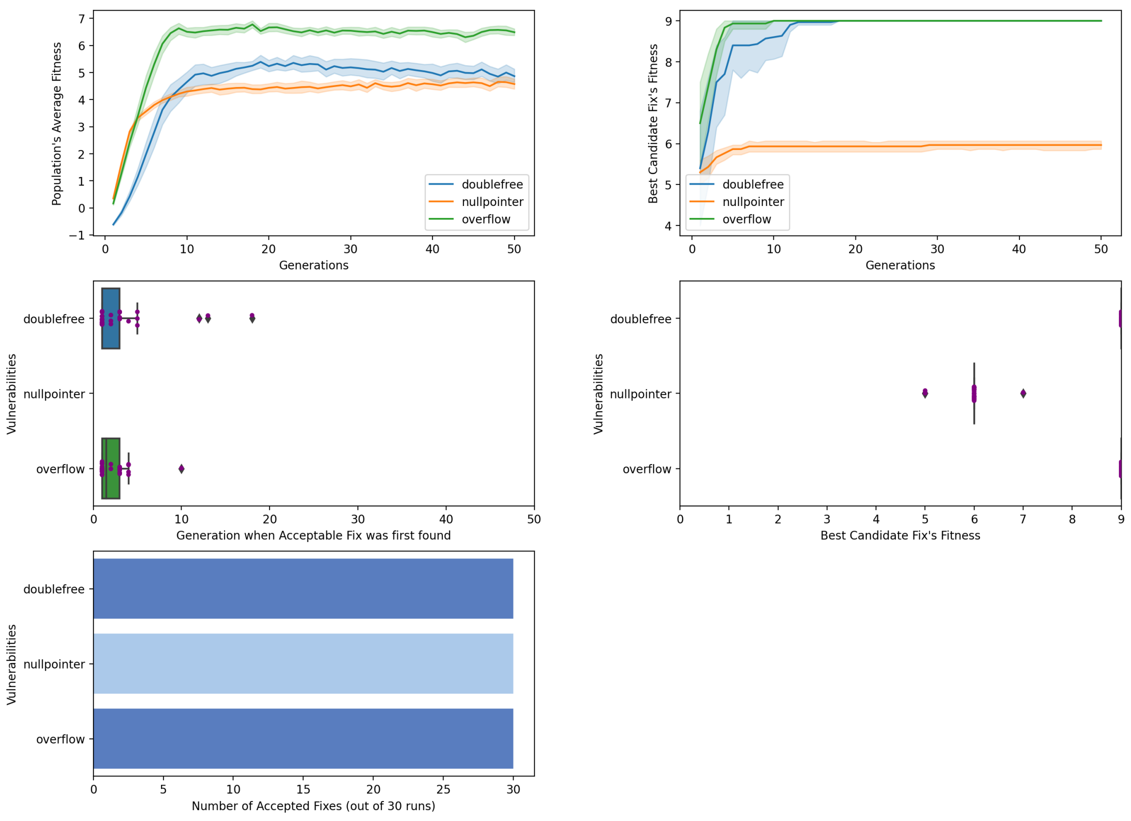


Figure B.7: trie pop100altlocs

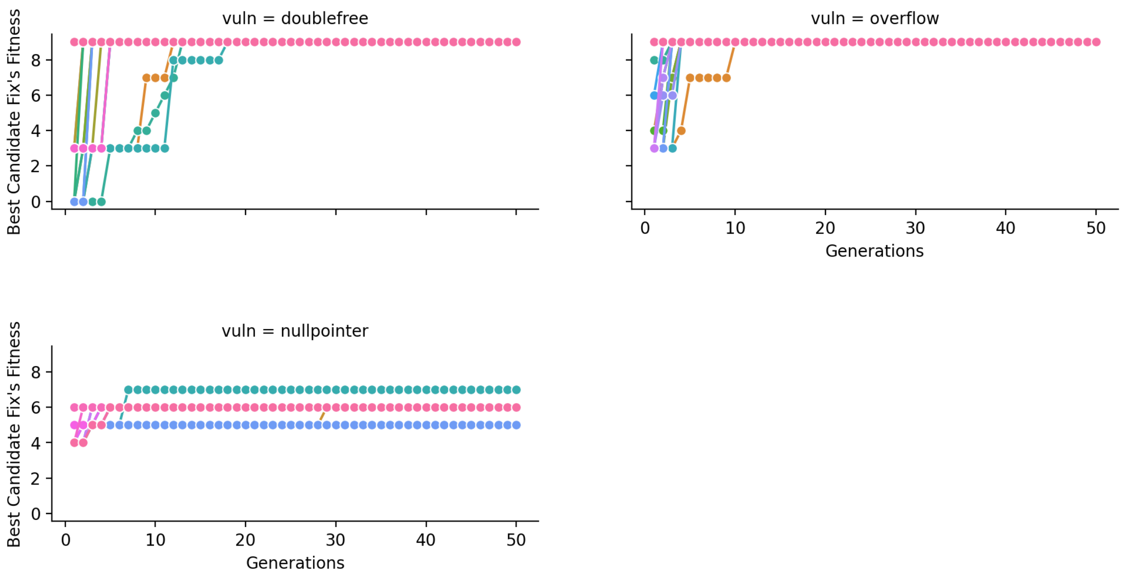


Figure B.8: trie pop100altlocs

B.4.5 Dynamic Fitness

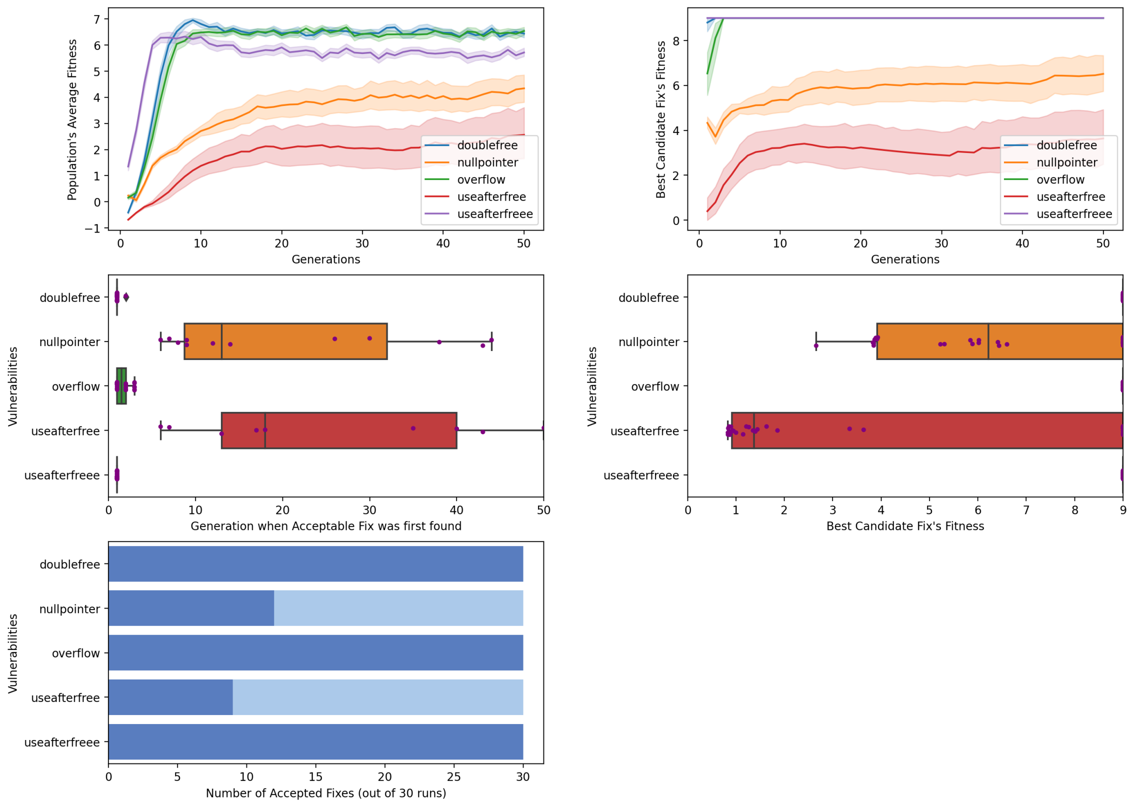


Figure B.9: trie pop100dynamicfitness

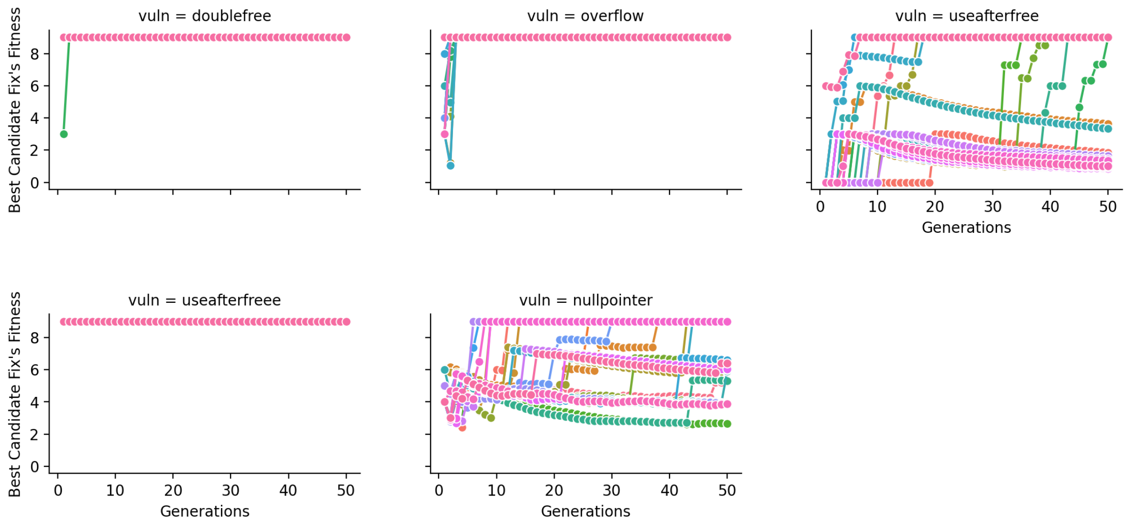


Figure B.10: trie pop100dynamicfitness