TÉCNICO LISBOA

A Comparative Study of Automatic Software Repair Techniques for Security Faults

Eduard Costel Pinconschi

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Rui Filipe Lima Maranhão de Abreu
Prof. Pedro Miguel dos Santos Alves Madeira Adão

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu
Member of the Committee: Prof. Miguel Nuno Dias Alves Pupo Correia

January 2021

To my family
— for all the care, support, and encouragement
throughout my life.

Abstract

The rapid expansion of Information and Communication Technologies (ICT) brought with it at the same pace cyber threats through the medium of security faults in software [1]. Fixing these require expert knowledge and a thorough understanding of systems. Within the field of software engineering, plenty of research has been made [2–4], proposing repair tools and techniques to increase the quality, productivity and reduce human intervention within the activities of debugging and testing. However, these must reason about significantly more security faults and give greater insight to human analysts [5]. We analyze the studies in the area to find these did not explore enough security faults nor a common baseline standard for evaluating them. That motivates a system for assessing, evaluating, and comparing the ability of Automatic Program Repair (APR) tools. With that system, we conduct the first case study to investigate the repairability, specificity, and suitability of APR tools to fix security faults. Our findings show that current state-of-art repair techniques can repair specific categories of security faults. However, these are farther from being effective as the subject tools fix individually at most 21.4%, and together 30.4% of 56 vulnerable applications which are among the top 25 most dangerous software weaknesses, such as *CWE-664*, *CWE-118*, and *CWE-682*. Also, we found that state-of-art tools can generate patches that fix the security fault but mostly compromise a program's functionality. As for the repair technique, genetic programming is more effective, and the data-driven is more efficient for repairing security faults.

Keywords

Gauges; Technology; Behavior; Automatic Program Repair; Security;

Resumo

A rápida expansão das Tecnologias da Informação (TIC) e Comunicação trouxe consigo ameaças cibernéticas por meio de falhas de segurança no software [1]. Corrigir essas falhas requer conhecimento especializado e uma compreensão completa dos sistemas. Na área da engenharia de software, muitos estudos têm sido feitos [2–4], propondo ferramentas e técnicas de reparo para aumentar a qualidade, produtividade e reduzir a intervenção humana nas atividades de depuração e teste. No entanto, devem tratar de significativamente mais falhas de segurança e fornecer uma melhor percepção aos analistas humanos [5]. Analisamos os estudos na área e descobrimos que as falhas de segurança são pouco abordadas. Motivados pela lacuna, desenvolvemos um sistema para avaliar e comparar a capacidade das ferramentas de reparo. Com esse sistema, conduzimos o primeiro estudo onde investigamos a capacidade de reparo, especificidade e adequação das ferramentas de reparo para corrigir falhas de segurança. Os nossos resultados mostram que as técnicas de reparo podem corrigir falhas específicas de segurança. No entanto, estão longe de ser eficazes, e essas corrigem individualmente no máximo 21,4% e em conjunto 30,4% dos 56 programas vulneráveis avaliados. As vulnerabilidades dos programas são classificadas no top 25 das falhas mais críticas em software. As principais classes são *CWE-664*, *CWE-118* e *CWE-682*. Além disso, descobrimos que ferramentas podem gerar patches que corrige a falha de segurança mas que comprometem bastante a funcionalidade do programa. Quanto à técnica de reparo, a programação genética é mais eficaz, e a baseada em dados é mais eficiente para reparar falhas de segurança.

Palavras Chave

Medidores; Tecnologia; Comportamento; Reparo Automático de Programas; Segurança;

Acknowledgments

First, I want to thank my mother and sister for always being there for me. Also, I am very grateful to my grandmother and father for their understanding and support during the last months of this work. To the rest of my family, thank you all for being with me.

I would also like to acknowledge and give credits to my supervisors, Prof. Pedro Adão and Prof. Rui Maranhão, for their guidance, insight, and support throughout my decisions. Particularly for always being active and very keen to help me with anything in their reach. Thank you for making this Thesis possible.

I would also like to thank the colleagues that helped me with parts of this work. Nuno Sabino, Sofia Reis, and Thomas Durieux – Thank you for your suggestion and your kind help.

Last but not least, I want to thank all my friends and colleagues for being there for me during the good and bad times in my life. Especially to Alexandre Borges and Amândio Faustino, who have been like brothers throughout my academic path. Thank you.

To every one of you – Thank you.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

AI	Artificial Intelligence
APR	Automatic Program Repair
ASR	Automatic Software Repair
AST	Abstract Syntax Tree
CIA	Confidentiality, Integrity and Availability
CGC	Cyber Grand Challenge
CRS	Cyber Reasoning System
CTF	Capture the Flag
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DARPA	Defense Advanced Research Projects Agency
FL	Fault Localization
ICT	Information and Communication Technologies
IoT	Internet of Things
LSTM	Long Short-Term Memory
ML	Machine Learning
NMT	Neural Machine Translation
NLP	Natural Language Processing
NVD	National Vulnerability Database
POV	Proof of Vulnerability
URL	Uniform Resource Locator

1

Introduction

Contents

Since the emergence of the Internet in the 1980s, interconnected computer networks have reached a global scale, connecting society through a wide variety of devices. Today's trends feature technologies to improve both cognitive and physical parts of the human body — e.g., prosthetic limbs and biochips [14]. Although the Internet of Things (IoT) being in its infancy, it's expanding rapidly and promises a future with smarter and better cities. Information and Communication Technologies (ICT) are becoming increasingly part of our lives and even our organism. We are dealing with a complex domain, cyberspace, deeply embedded in our surroundings, and in a permanent state of risk.

The rapid expansion of ICT, brought with it at the same pace cyber threats through the medium of security faults in software. The Common Vulnerabilities and Exposures (CVE) database [1] lists over 146,400 records of security vulnerabilities. These originate from exploitable faults in the software, allowing malicious individuals to perform unauthorized actions in a system. A common attack vector is malware, malicious software that takes advantage of security vulnerabilities to cause harm and spread to new victims by the same means [15]. These can take the form of advanced persistent threats to cause serious damage by sabotaging systems, as Stuxnet [16] did in the Iranian nuclear power station.

Merely manual approaches performed by human experts can not cope with the increasing quantity, complexity, and variation of vulnerabilities. These require expert knowledge, a thorough understanding of systems, and large amounts of time and effort invested when performed manually. Not to be forgotten the window of exposure in the affected system. Within the field of software engineering, plenty of research has been made [2–4], proposing repair tools and techniques to increase the quality, productivity and reduce human intervention within the activities of debugging and testing. These solutions are far from removing the need for an in-depth understanding of systems and expert assistance, and even further, from fixing security faults.

A leaping step towards the automation of cyber defense was fostered by Defense Advanced Research Projects Agency (DARPA) with Cyber Grand Challenge (CGC) in 2014-2016. The tournament pitted computers against each other to explore the potential of automatic discovery and correction of vulnerabilities in systems. DARPA's CGC successfully bridged the gap between software and system testing technologies with cybersecurity. Recent advancements in the field of Artificial Intelligence (AI) bring promising technologies with the potential to be integrated into a fully automated system capable of discovering and patching a wide variety of security faults.

Within the immense body of published work relating to Automatic Software Repair (ASR), the corpus of literature relating to AI techniques have started to gain shape, and the literature related to cybersecurity is rudimentary. ASR techniques must be able to reason about significantly more types of security faults and to give a greater insight over the addressed security faults to human analysts [5]. These security faults have many dimensions to be considered — e.g., programming languages, technology, behavior.

1.1 The Problem

Program repair bots [17] have demonstrated their potential in the industry for identifying and repairing faults in programs. These are now generalized to the repair of critical faults such as security vulnerabilities [18–24]. However, no study has evaluated and compared their ability to repair security faults. That leads us to the open challenge of evaluating with a common baseline Automatic Program Repair (APR) tools in a security context. Therefore, to achieve that, the following issues need to be solved:

Issue 1: past studies did not consider a method focused on security aspects that allow for the assessment, evaluation, and comparison of repair techniques.

Issue 2: past studies techniques did not evaluate a set of programs that transmute and stress the wide variety of security faults.

Issue 3: past studies did not leverage a common language or taxonomy to describe and discuss their ability to patch security faults.

Issue 4: past studies provide negligible insight into the performance of repair techniques with respect to common metrics that allow pointing differences that reveal their suitability.

1.2 Contributions

Considering the issues listed above, this dissertation has the following contributions:

A framework to assess, evaluate, and compare repair techniques concerning security faults. The solution has the name *SecureThemAll*, and Section 4.3 describes its implementation;

Adaptation of a benchmark with 56 vulnerable programs to support APR techniques. The solution has the name *cb-repair*, and Section 4.4 describes its implementation;

Adaptation of a data-driven approach into a test-based tool for fixing security faults. The approach has the name *CquenceR*, and Section 5.2 describes its implementation;

A curated dataset of security vulnerabilities in C language for training data-driven approaches. Section 5.3 describes the data selection, collection, and preparation;

A quantitative comparative study on 3 state-of-the-art repair techniques, evaluating their ability to fix specific classes of security faults.

1.3 Organization of the Document

This thesis is structured as follows:

Chapter 2 defines the fundamental concepts used throughout this dissertation, summarizes and structures the body of work related to automatic software repair, introduces the Cyber Reasoning System (CRS), and provides an overview on the state-of-the-art data-driven approaches.

Chapter 3 formulates the context around security faults and describes the techniques for repairing security faults.

Chapter 4 presents the research questions, points out the open challenges, describes the proposed system, explains the underlying design decisions and the consideration made during the selection of resources.

Chapter 5 presents CquenceR, an adaptation of a data-driven approach to patch security faults, along with its implementation details, data construction and analysis, and evaluation.

Chapter 6 reports the results gathered from the comparative study with particular focus on the questions that drive the problem and presents the main findings.

Chapter 7 reflects on the main contributions of this dissertation, presents the limitations, and provides discussion on the future work.

2

Fundamental Concepts and Related Work

Contents

This chapter introduces the core concepts, provides an overview of the techniques related to the problem addressed in this dissertation, and demonstrates how it's scope goes beyond the academic context. In section 2.1 is described and organized the automatic software repair literature. Then in section 2.2 is introduced the first technology for automated cyber defense. Finally, in sections 2.3 are presented the recent studies that explore promising data-driven technologies.

2.1 Automatic Software Repair

M. Monperrus [25] organizes the body of knowledge into two families of repair techniques, behavior and state repair. L. Gazzola et al. [3] introduce a comprehensive survey that names these technologies as software repair and software healing, respectively. Further, the survey illustrates a conceptual framework that covers the commonalities and differences of the technologies. Software repair techniques locate faults and change a program's behavior by altering its code with corrective adjustments that fix the faults. Software healing consists of changing a program's state to restore its normal execution and often mitigates software faults or prevents failures. Thus, the terms of behavior repair and state repair are interchangeable with software repair and software healing. The term APR is widely used throughout the literature to refer to the family of behavior repair techniques, and we stick with it. Software healing technology repairs programs by responding to failures. That's beyond our scope, as patching security faults revolves around fixing source code faults. Within the behavioral repair domain, Y. Liu et al. [26] introduce a taxonomy for the many repair techniques using test-suites as specifications, coined as Test-Based Repair.

2.1.1 Terminology

This subsection introduces the core terms used within software repair literature, and throughout the following chapters. Although the terms have several synonyms in the literature, these maintain the clarity of the core concept. The definition for the terms below are borrowed from [25].

Definition 2.1.1 (Failure)**.** *"A failure is an observed unacceptable behavior of a program execution"*.

Definition 2.1.2 (Error)**.** *"An error is the incorrect state prior to the failure, propagating without yet having been noticed"*.

Definition 2.1.3 (Fault)**.** *"A fault is the root cause of the error, in the incorrect code"*.

Definition 2.1.4 (Bug)**.** *"A bug is a deviation between the expected behavior of a program execution and what it actually happened"*.

Remark. The term bug is interchangeable with terms of fault and defect.

Definition 2.1.5 (Specification)**.** *"A specification is a set of expected behaviors"*.

Definition 2.1.6 (Test-suite)**.** *"A test suite is an input-output based specification"*.

Remark. The term test-suite is interchangeable with term oracle, since is the most used validation method.

Definition 2.1.7 (Automatic Repair)**.** *"Automatic repair is the transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification"*. The repair is performed on the source code.

Remark. The term repair is interchangeable with terms of patch and fix.

2.1.2 Automatic Repair Process

Overall, the automatic repair process consists of collecting information from a program's correct and faulty executions. That's used to identify the faulty locations and apply the necessary corrective changes through algorithms. The correctness of the changes is proven when the program executes all cases from the test suite correctly. The process assembles through the activities of localization, repair, and validation, as described in [3]. Below we define these based on that.

Localization has the purpose of identifying possible faulty locations in code by employing techniques that rank statements according to their relationship with the fault. Automatic repair techniques generally adapt for this step Fault Localization (FL), section 2.1.3

Repair consists of replacing the faulty statements with possible corrective changes, namely candidate patches. Those are generated with algorithms that leverage internal/external information to the program. Such approaches are distinguished by classes, depending on the way the repair is defined and addressed, section 2.1.4

Validation considers a specification to determine whether the generated candidate patches repair the fault and maintain the program's functionality. A vast amount of repair techniques make use of a test suite as a type of specification. Tests allow obtaining assertive information such as expected output from the program's execution.

2.1.3 Fault Localization

FL techniques serve the purpose of guiding the activity of program debugging by identifying the location of faults. W. Wong et al. [27] present a comprehensive overview of FL techniques and discuss the key issues. These are classified into eight categories, from which spectrum-based is one of the most dominant approaches, and commonly used by repair techniques, hence, here briefly described. Spectrum-based FL (SBFL) techniques use a program's execution information to identify suspicious code, such as the

similarity coefficient-based approach. It uses a formula over the coverage results to compute the suspiciousness value of each statement. The coverage comes from the execution of passing and failing tests.

2.1.4 Repair Techniques

In [3] the repair techniques are categorized into generate-and-validate fig. 2.1 and semantics-driven fig. 2.2. The former defines a search space which is explored for potential solutions, while the latter represents the problem of repairing a program as a formula or procedure, whose solutions might be found with constraint solving or program synthesis, respectively. Generate-and-validate techniques produce candidate patches through change operators classified as atomic changes, pre-defined and example-based templates, as described in [3]. Below we define these based on that.

Atomic Changes modify a program in a single point with insert, delete or modify statements.
Pre-defined templates modify a program with pre-defined patterns.
Example-based templates modify a program with systematic changes or evolving techniques extracted from historical data.

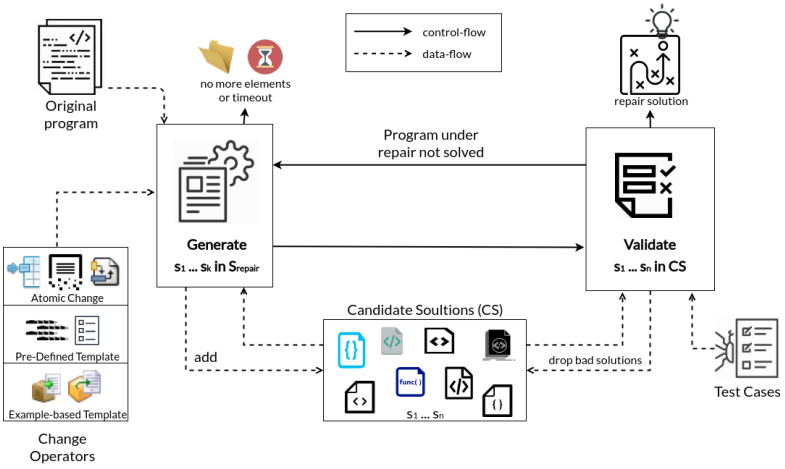


Figure 2.1: Generate-and-validate repair process (*cf.* [3]).

2.1.5 Validation Methods

As described in section 2.1.2, the correction of the changes applied in the repair process is proven based on a specification. Beside test-suite, other types of specifications, less used, are reference implementation [2], design-by-contract and abstract behavioral models [25]. The former can make use of a program that implements a specification that demonstrates the correct behavior. Design-by-contract

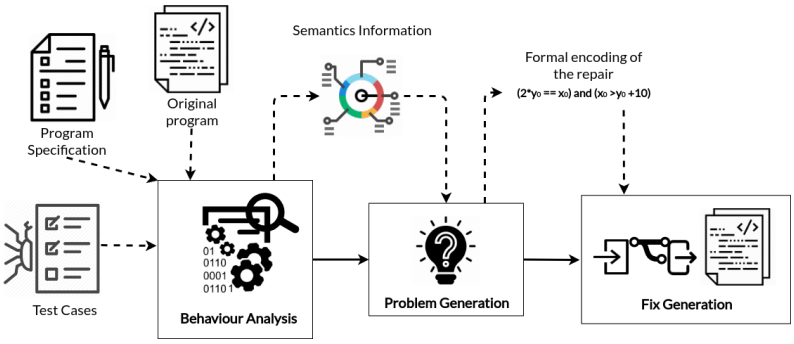


Figure 2.2: Semantic-driven repair process (*cf.* [3]).

dictates ”what a program does” with preconditions, and ”how it does” with postconditions, for computing the possible program states. Abstract behavioral models represent the program with a model that captures its behavior, such as a state machine.

2.1.6 Faults Design

As described in [3], the design of repair techniques concerning faults can be general or fault-specific, meaning their ability to address software faults can be broader or narrower. Thus, we use the term fault design to refer to the software faults a repair technique targets. The notorious buffer overflow vulnerability is a specific fault that results from simple errors in unsafe languages, such as C. A. Smirnov et al. propose PASAN [28], a compiler extension able to generate patches that detect and seal buffer overflows attacks. The authors suggest the tool can be generalized to accommodate a broader scope of attacks.

Representative repair techniques with general fault design can repair specific faults. The popular generate-and-validate tool, GenProg [29] combines Genetic Programming (GP) with atomic change operators to repair automatically generic faults in C code. The tool can cover classes of vulnerabilities, such as buffer overflows and format strings. In the category of semantic-driven repair, S. Mechtaev et al. propose the tool Angelix [30], which synthesizes repairs based on semantic information of the program. The tool can repair dependent multiple buggy locations and is the first work that reports the automated repair on the well-known Heartbleed vulnerability.

2.1.7 Root Cause of Faults

As described in section 2.1.4, one way of approaching the automatic repair process is to get to the root cause of faults, as done by some generate-and-validate techniques that use template operators. That points out another paradigm, one where repair tools are categorized by their ability to fix faults. M. Monperrus similarly organized the literature on automatic software repair into a living review [2]. Most

repair techniques target dynamic errors, errors raised during run time, and static errors such as syntax errors. The rest of the repair techniques are grouped into targeted repairs, such as tests, concurrency errors, build scripts, web applications, software models, and security vulnerabilities. However, that's not the same as targeting common root causes of faults. Gemma et al. [31] give a better understanding of that with a taxonomy for faults, resulting from the analysis of a large corpus of existing bug reports. Their taxonomy characterizes faults into nine common root causes:

> (A) Configuration issue; (B) Network issue; (C) Database-related issue; (D) GUI-related issue; (E) Performance issue; (F) Permission/Deprecation issue; (G) Security issue; (H) Program Anomaly issue; (I) Test Code-related issue.

2.1.8 Empirical studies

M. Martinez et al. [32] perform the first experiment on evaluating test-suite based repair on a benchmark of real-world Java bugs, assessing the real correctness of patches that pass the test suite. Y. Jooyong et al. [33] conduct the first correlation study of APR with traditional metrics used in software testing — statement coverage, branch coverage, test-suite size, and mutation score. The repair tools, GenProg and SemFix, were evaluated on ten subject C programs. The results imply the metrics are adaptable to APR, and the reliability of generated repairs proportionally increases with the number of metrics. X. Kong et al. [34] investigate the impact of five representative repair techniques on general APR, regarding the repair effectiveness and efficiency. The experiment considers 17 subject C programs and measures the output with three metrics — success rate, false-positive rate, and repair execution's CPU time. M. Tufano et al. [35] validated through an empirical study the feasibility to patch defects in Java methods via Neural Machine Translation (NMT) techniques trained with bug-fixing patches in the wild. K. Liu et al. [36] conduct a critical review on the evaluation of 11 APR tools' repair on Java programs and propose eight evaluation metrics regarding the performance for reducing the comparison biases. These metrics cover the impact of the used FL, patch generation limitations and efficiency, intrinsic attributes of bugs, and the benchmark over-fitting.

2.2 Cyber Reasoning Systems

Throughout the years, research scholars introduced related techniques for handling security faults in software — i.e., vulnerability detection [37], exploitation [38, 39], and patching. With the attempt to reduce human effort, these became progressively automated, and their separable use for defending software came together, pointing to a unified technology. That has the name of CRS, an automated sys-

tem capable of performing vulnerability detection, exploit generation, and software patching to defend software [40]. To that end, DARPA put out a call to the experts in computer security through the means of a competition.

Vulnerability detection — software analysis performed for detecting the presence of vulnerable statements in software.
Exploit generation — programs or data crafted to trigger the vulnerable elements in code that enable the undesirable behaviour of a security fault.

2.2.1 DARPA'S Cyber Grand Challenge

To pave the way towards a future where software safety is the expert domain of machines, DARPA launched the CGC, the world's first CRSs competition [41]. The competition was held through two events, the CGC Qualifying Event in 2015 and the CGC Final Event in 2016. DARPA's CGC challenged highly trained experts to have their autonomous systems compete against each other on a Capture the Flag (CTF) tournament circuit to defend a set of software services.

Services — instances of applications purposely designed and build as network services, each containing at least one memory corruption security fault, for challenging autonomous vulnerability discovery and remediation systems [7]. These are named *challenges* and have two unique properties, exclusivity — i.e., excludes analogy with other applications — and singularity — i.e., restrains the context to the competition, without any real-world impact — of security faults.

Competition Model

In a CTF tournament, contestants discover, exploit, and fix security faults in their services in real-time as in a battle against their adversaries. CGC's contestants were fully autonomous CRSs responsible for addressing the discovery, proof, and mitigation of software security faults.

A CRS scored points by protecting and keeping its services running and by exploiting adversaries' services. A CRS lost points by having their systems exploited or by damaging their services. Therefore, the game was centered around two activities, defense, and offense of vulnerable software in services. Below we briefly explain these activities based on [42].

Defense — CRS had to rely on intrusion detection and binary patching. The former was performed by writing and deploying firewall rules, and the latter by finding and patching vulnerabilities in the services at the machine-code level.
Offense — CRS had to prove the existence of a security fault in the software of the target service, with a program that either executed code within the target or leaked it's privileged data

Research Challenges

The CGC dared the contestants with the challenge of identifying, proving, and patching security faults in unseen binary code, without disrupting functionality [42]. That required a holistic approach, through fully autonomous systems, and at the speed of seconds or minutes. Contestants overcame those challenges with techniques central to cybersecurity, achieving full automation of cyber defense and offense. DARPA's CGC success validates the concept of automated cyber defense. The first generation of CRSs bridge the gap between software security and program analysis research. Meanwhile, as pointed in the related literature [40, 43], there are many open challenges, and the astounding development of AI brings new opportunities for addressing the further improvements of CRSs.

2.3 Data-driven Software Repair

In recent years, advancements in AI brought a wave of progress within the fields of software engineering. Particularly for programming languages, from research exploring the applications of probabilistic models [44]. This trend gained thrust as well with automatic software repair, coined as data-driven [2, 45]. The approaches connect prior knowledge to a large code corpus, rather than the traditional techniques based on manually written rules resulting from intuition or expert experience.

The goal of AI is to produce intelligent agents with the cognitive ability to solve problems, make decisions, and learn from the environment. Before diving into the data-driven repair, we give a brief description based on [44, 46] of the subsets of AI commonly used for data-driven software repair, namely Machine Learning (ML) and Natural Language Processing (NLP).

> **Machine Learning** - is about making data-driven decisions, that is, decisions based on learning from data. It consists of a series of methods, models, and algorithms used for data analysis. The data defines the quality of results even more than the choice of the algorithms employed. In ML patterns are identified in data to make predictions. All that matters are the input data and output results, functioning as a black box.
>
> **Natural Language Processing** — bridges natural languages and computers by aiding machines to understand, process, and analyze human language. A relevant core concept in NLP is sequence-to-sequence (seq2seq), a model representing both input and output as a sequence. Examples of applications for these models are machine translation, text summarization, speech-to-text, and text-to-speech. NLP methods are increasingly reliant on data-driven approaches. [46].

Both ML and NLP involve *supervised* or *unsupervised* learning. Supervised learning uses labeled data to learn general rules by matching input with output information and predict new unseen data. Unsupervised learning makes sense of unstructured data without supervisory signals and tries to identify implicit patterns, structures, and relationships within the data set.

2.3.1 Repair Techniques

F. Long and M. Rinard [47] introduce the first data-driven repair tool, Prophet. The approach has the insight that correct code properties are shareable across applications and can be leveraged to fix incorrect applications. The approach uses a probabilistic and application-independent model of correct code, trained to generate a search space of candidate patches which are prioritized in order of likely correctness. During the same year, D. Le et al. [48] come with the intuition that program repair can be guided with derived patterns from bug fixes from development history of software projects. Genesis tool [49] implements the same core idea of using common fix patterns by abstracting specific details from multiple patches, namely patch generalization as introduced by F. Long et al. That is further combined with a search space inference algorithm to generate a search space of candidate patches. Delvin et al. [50] address the repair of four classes of common Python bugs, with a system that learns from a large corpus of real-world source-code to predict both bug locations and fixes, without any information about the correct functionality of the program. In their approach is used a neural neural network to encode the AST input and a specialized network module to scores each repair candidate. Daniella et al. [45] frame the problem of finding and fixing bugs in Javascript programs as a structured prediction problem on a graph-based representation of programs. The approach targets a wide range of bug types, and is implemented into a tool named HOPPITY. Z. Yu et al. [51] propose and approach to learn the relation between code and edits by representing code as Abstract Syntax Tree (AST) annotated with information and edits as code transforms — i.e., changes on the AST structure. The system takes as input a program's code and predicts using structured prediction the most likely edits to be applied.

NMT-based Repair

Y. Pu et al. [52] follow the intuition that frequent and similar code fragments across programs can be re-used for automatic program correction. For that, a neural network model used in NLP is modified and trained to recover correct fragments from a large corpus of programming assignments. Those are used to replace incorrect fragments to generate candidate fixes. A system called sk_p implements the approach, and shows decent ability of correcting both syntactic and semantic errors in small programming assignments in Python. Hata et al. developed [11] Ratchet, an NMT-based technique following the idea of corrective patch generation by learning code hunks from past fixes—the pairs of pre-correction and post-correction statements that are contiguous. Ratchet focuses on single-statement changes within Java methods, and performs better than pattern-based patch suggestion. That demonstrates NMT-based learning approaches have the ability to address the issue of automatic patch generation. Although the promising results, NMT approaches suffer from out-of-vocabulary—rare words used infrequently in specific contexts are not included in the vocabulary. The vocabulary challenge is overcome by Chen et al. in [9] with a copy mechanism that allows to capture the rare words from the input. The technique is implemented along with an abstract buggy context—for capturing long-range dependencies—into a

program repair tool called SEQUENCER. The tool is based on sequence-to-sequence learning and produces well formed one line fixes for one line bugs for Java programs. The same technique is used with a different approach to handle the vocabulary problem in [23] by Chen et al. to predict patches for vulnerabilities in C code from the CVE database. L. Thibaud et al. [53] combine convolutional neural networks (CNNs) and a context-aware NMT architecture to automatically fix bugs in multiple programming languages. The approach separates the faulty lines from the context, and combines models to capture different relations between faulty and clean code. The approach fixes 509 bugs over six bug benchmarks in four programming languages.

Table 2.1: Data-driven Repair Techniques

Study/Repair Tool	Probabilistic Model	Programming Languages
Prophet [47]	Parameterized likelihood function assigns each candidate patch a probability	C
sk_p [52]	seq2seq neural network model	Python
History Driven Program Repair [48]	Graph-based representation of bug fixes and stochastic search algorithm as optimization	Java
Genesis [49]	applies a generalization algorithm on subsets to infer a transform, and formulates and solves an integer linear program	Java
SSC model [50]	we encode the AST with a sequential bidirectional LSTM	Python
Ratchet [11]	NMT-based technique tokenizes statements uses a single layer of LSTM cells	Java
SEQUENCER [9]	NMT-based technique tokenizes statements sequence-to-sequence model	Java
Using Sequence-to-Sequence Learning for Repairing C Vulnerabilities [23]	NMT-based technique. Tokenizes statements. Sequence-to-sequence model	C
HOPPITY [45]	parse code into an AST represents programs using graphs use a Graph Neural Network	JavaScript
Learning the Relation between Code Features and Code Transforms with Structured Prediction [51]	Code represented as AST nodes. Uses conditional random field, appropriate for tree-based data	Java
CoCoNuT [53]	Tuples of buggy, context, and fixed lines. Input as sequences of tokens. NMT model uses CNN layers	Java, C, Python, and JavaScript

3

Background

Contents

This chapter builds the background for the problems addressed in this dissertation. Section 3.1 formulates the context and gives an insight into the root cause of security faults.

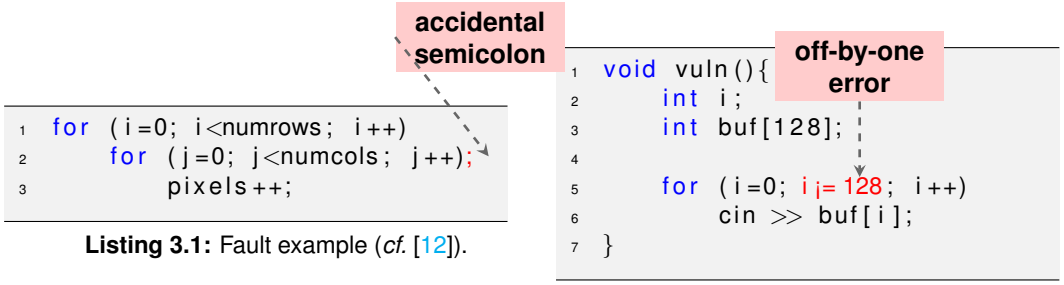
3.1 Context Formulation

The previous chapter gives an overview of automatic software repair, shows its scope in the context of cyber defense, and describes state-of-art techniques using AI technologies. This section elaborates the focus of the dissertation within the context of security. It begins by describing security faults followed by a review of the related automatic repair techniques.

3.1.1 Security Faults

As discussed in section 2.1.7, software faults have several root causes, and security faults are among them. Although the same defect nature, it's important to have under consideration what originates a security fault. The main difference is that faults cause expected scenarios not to run, while security faults leave an open window for violating the Confidentiality, Integrity and Availability (CIA) of a system. The example below depicts the difference. The fault in listing 3.1 makes the *pixels* variable to not be incremented. The security fault in listing 3.2 results from the wrong bounds-check in the *for* cycle, making the system susceptible to a memory error. The error can be exploited by loading shell-code into the allocated buffer memory, leaving the system exposed to potential malicious use cases. A security fault that is accessible and exploitable is a vulnerability.

Remark. The term security fault is interchangeable with terms software weakness and vulnerability.




```
1 for (i=0; i<numrows; i++)
2     for (j=0; j<numcols; j++);
3         pixels++;
```

Listing 3.1: Fault example (*cf.* [12]).

```
1 void vuln(){
2     int i;
3     int buf[128];
4
5     for (i=0; i¡= 128; i++)
6         cin >> buf[i];
7 }
```

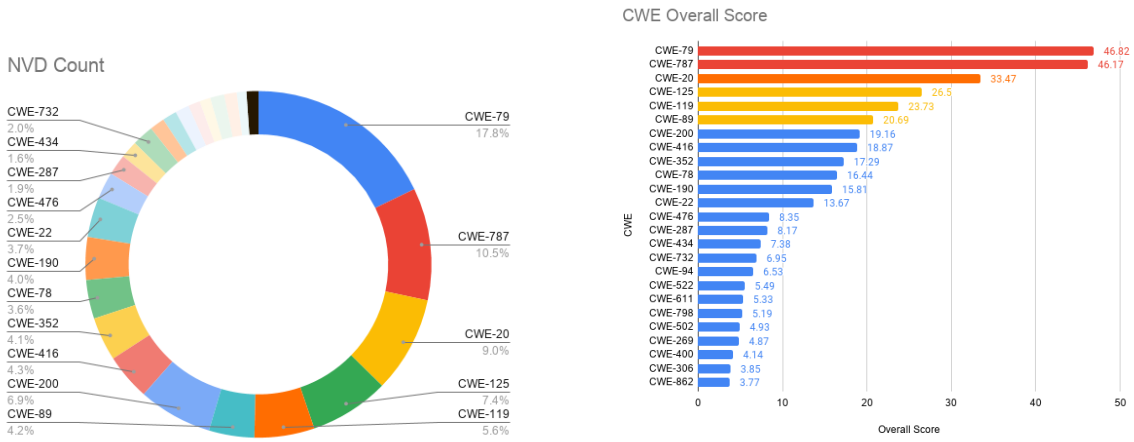
Listing 3.2: Security fault example (*cf.* [13]).

Figure 3.1: Difference between a security fault and a security fault.

Types

The MITRE Corporation [54] uses the term "weaknesses" to cover a broad spectrum of security faults that makes both hardware and software susceptible to unauthorized actions that can be performed by

an attacker. To discuss security issues in a common language, MITRE developed a list of common weakness types, namely Common Weakness Enumeration (CWE). The latest view of software weaknesses [55] contains 419 types of weaknesses encountered in software development.



(a) Number of entries related to a particular CWE within the NVD data set (*cf.* [56]).

(b) Weaknesses score based on prevalence and severity (*cf.* [56]).

Figure 3.2: 2020 CWE Top 25 Most Dangerous Software Weaknesses.

In fig. 3.2 are listed the top 25 most common and impactful software security faults experienced over the years 2018 and 2019. According to MITRE, these security faults "are dangerous because they are often easy to find, exploit, and can allow adversaries to completely take over a system" [56]. The list was developed by leveraging published vulnerability data from National Vulnerability Database (NVD), consisting of approximately 27,000 CVEs, each associated with a weakness.

Common Vulnerabilities and Exposures CVE — standardized description serving as one identifier for one vulnerability or exposure [1]

The doughnut chart in fig. 3.2(a) shows that the top 5 weaknesses in the list account for more than 50% of the total CVEs in the data. The bar chart in fig. 3.2(b) shows the overall score associated with the weaknesses—the numerical score represents their severity and is calculated based upon a standardized set of characteristics. A high score was given to common weaknesses that can cause significant harm. The following are the top 6 most severe weaknesses, with a score greater than 20:

1. **CWE-79**: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting');
2. **CWE-787**: Out-of-bounds Write;
3. **CWE-20**: Improper Input Validation;

4. **CWE-125**: Out-of-bounds Read;
5. **CWE-119**: Improper Restriction of Operations within the Bounds of a Memory Buffer;
6. **CWE-89**: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

Vulnerable Programming Languages

The latest Common Weakness Enumeration (CWE) list [6] contains additional helpful views with CWEs that cover issues found in a programming language—C, C++, Java, and PHP—that are not common to all languages. The bar chart in fig. 3.3 shows the number of CWEs by programming language with issues uncommon to all languages. Additionally, those have been mapped to the CWEs in top 25 fig. 3.2. According to that, C++ has the most uncommon issues to all languages, followed by C. Both have the same CWEs from top 25—*CWE-787, CWE-125, CWE-119, CWE-416, CWE-476*. Three of them are in the top 6 most severe weaknesses. CWE-502 from the top 25 is in both Java and PHP.

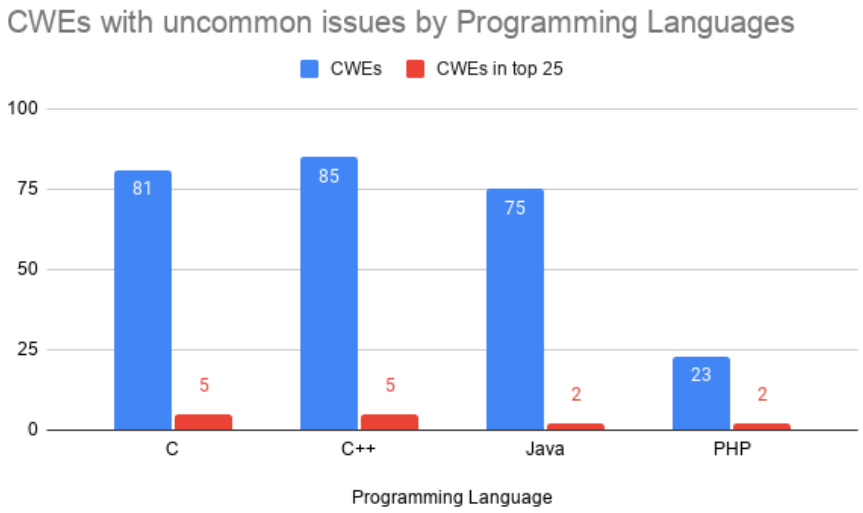


Figure 3.3: Number of CWEs by programming language that cover issues uncommon to all languages (*cf.* [6]).

3.1.2 Repair of Security Faults

As described in section 2.1.6, the same concept applies to repair tools that attempt to repair security faults, their design can be general or specific.

Specific techniques

The techniques cover security faults such as buffer overflows, component hijacking, cryptographic misues, and memory leaks. F. Gao et al. [18] propose BovInspector, a method that inspects static warnings

to detect, validate, and repair buffer overflows. The tool can fix six C programs with buffer overflow faults, proposing patches similar to the official repair. In [19] M. Zhang et al. introduce the first approach for automatic patch generation in Android applications, into a prototype called AppSealer that targets component hijacking vulnerabilities. The technique tracks dangerous information flows, faults that compromise the data integrity by leaking and changing sensitive information without the related permissions. In their evaluation, AppSealer was able to mitigate 16 vulnerabilities from SQL injection and delegation attack threats.

S.Ma et al. [20] propose CDRep to repair seven cryptographic misuse defects—CWE-310 Cryptographic Issues—in Android applications. The approach applies transformations on matching placeholders from manually created patch templates—by generalizing cryptographic misuse fix examples. In their experimentation results, the tool was able to repair the 94.5% of 1262 vulnerable apps. Q. Gao et al. [21] propose LeakFix, a semantic-driven tool for fixing memory leaks. The approach fixes leaks in code by inserting deallocation statements. In the evaluation, the tool fixed 25 out of 89 leaks in 15 C programs.

General techniques

In [24] Ma S. et al. propose VuRLE, a tool for automatic detection and repair of multiple types of vulnerabilities in Java source code. VuRLE performs that by applying learned repair templates containing patterns extracted from previous fixes for vulnerable code. VuRLE was capable of repairing 101 out of 279 vulnerabilities. The types of vulnerabilities in VuRLE's dataset are:

1. **CWE-389**: Error Conditions, Return Values, Status Codes;
2. **CWE-772**: Missing Release of Resource after Effective Lifetime;
3. **CWE-1211**: Authentication Errors;
4. **CWE-310**: Cryptographic Issues;
5. **CWE-89**: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

J. Harer et al. [22] propose an adversarial learning approach that doesn't require paired data examples for fixing security vulnerabilities in C code. The approach is evaluated on SATE IV, a dataset with synthetic C/C++ vulnerable code examples across 116 different CWE types, achieving close results to a sequence-to-sequence learning baseline. Z. Chen et al. [23] propose sequence-to-sequence learning for repairing vulnerable functions in C code. The approach was evaluated on known CVE vulnerabilities from the NVD data feeds and can fix 14 out of 630 large vulnerable functions.

4

Proposed System

Contents

This chapter introduces the problem, describes the proposed solution, explains its underlying components, and justifies the design decisions. In section 4.1 are presented the research questions for the problem along with the open challenges. In section 4.2 is given an overview of the proposed solution along with the requirements. Then in section 4.3 is introduced the framework solution along with the considered subject repair tools. Finally, section 4.4 presents the benchmark solution along with the considered subject benchmarks.

4.1 Problem Definition

Chapter 2 introduced considerable related work and resources on automatic software repair delivered by researchers—such as tools, techniques, and empirical studies. The rise of new technologies brings hope for further developments of the next generation of systems to overcome most of the current limitations. In Chapter 3, the scope was focused on security faults, showing the progress made on the subject, and pointing to some uncharted territory — the variety of types and critical faults, uncommon issues specific to a programming language. Remarkably enough is general repair techniques for security faults use a data-driven approach, in contrast to specific ones. That shows repair techniques began to follow different approaches and consider more types of security faults.

4.1.1 Research Questions

The context in section 3.1 frames the current state of APR for security faults and positions it at the beginning of a long road of development towards a mature state. Positively, the following questions will take us closer to that destination.

Q1. [Repairability] Are the state-of-art repair tools able to patch security faults?

Q2. [Specificity] What categories of security faults are the state-of-art repair tools able to patch?

Q3. [Suitability] Which paradigm is the fittest for patching security faults?

4.1.2 The Problem

The previous questions and the comprehensive overview in chapter 2 reveals us the problem of **past studies not exploring enough security faults, nor a common baseline standard for evaluating them**. The problem is partitioned into the following issues and associated with the questions:

Issue 1: past studies did not consider a method focused on security aspects that allow for assessment, evaluation, and comparison of repair techniques — **Q1**;

Issue 2: past studies techniques did not evaluate a set of programs that transmute and stress the wide variety of security faults — **Q2**;

Issue 3: past studies did not leverage a common language or taxonomy to describe and discuss their ability to patch security faults — **Q2**;

Issue 4: past studies provide negligible insight into the performance of repair techniques with respect to common metrics that allow pointing differences that reveal their suitability — **Q3**;

4.1.3 Open Challenges

The previous issues along with the related work, point out to the following challenges:

C1. Proposal of a framework that allows for a the assessment, evaluation, and comparison of repair tools with respect to security faults - **I1**/**I3**;

C2. Proposal of a benchmark with a variety of vulnerable programs, and prepared for supporting APR tools - **I2**/**I3**;

C3. Proposal of repair techniques with a general fault design, targeting security faults - **I2**/**I3**;

C4. Proposal of a comparative study that evaluates with performance metrics the ability of repair tools to fix security faults. - **I4**;

4.2 The System

The challenges introduced in section 4.1.3 give us an idea of a solution suitable for answering the research questions and capable of tackling the issues in section 4.1.2. That is, **to develop a system that permits the assessment, evaluation, and comparison of APR tools' ability to fix programs with security faults and conduct a comparative study with it**.

We leverage previous work to borrow foundations, and the closes we found is the system proposed in [4]. The RepairThemAll framework runs repair tools on different benchmarks of bugs and collects the resulting patches for analysis. We follow the same paradigm, with the difference of the programming language, and fault design. Also, we propose comparative capabilities for the system. Thus, the system must execute repair tools on a benchmark with vulnerable programs and collect the generated results to evaluate and compare their ability. Figure 4.1 illustrates the architecture of the proposed system with it's paramount components.

Remark. The RepairThemAll framework considers Java program repair tools, thus, covers existent benchmarks for Java. In contrast, our scope is considerably different, as described throughout this chapter. We consider that and only leverage the design as a basis, rather than extending the RepairThemAll. Also, to avoid adding complexity from unnecessary functionalities and dependencies.

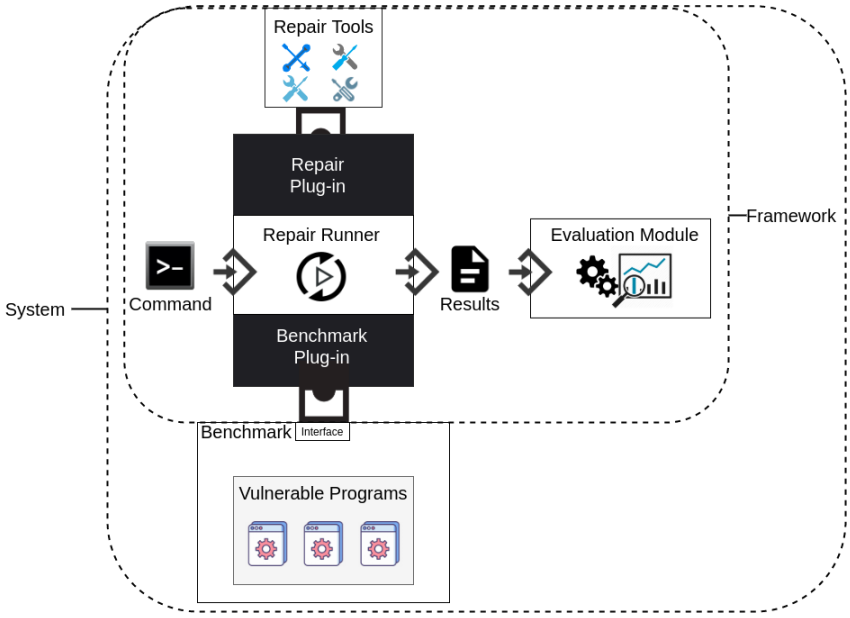


Figure 4.1: System Architecture (*cf.* [4]).

4.2.1 Abstractions

Taking into account the abstractions in RepairThemAll, the system should provide an abstraction around repair tools—*repair tool plug-in*, *repair runner*, and *evaluation module*—and benchmark—*benchmark plug-in*. Below we describe these abstractions based on [4].

Repair tool plug-in ought to abstract the common parameters that are required by repair tools, allow the addition and removal of tools, and process the generated results into a common format. Abstracting the parameters for the repair tools is necessary as these require different parameters. The common format should abstract the generated output and promote easy analysis and readability. Results should contain the textual difference between the generated candidate patches for the vulnerable program files.

Repair runner should allow for easy interaction with the system. Its role is to launch the execution of repair tools on specific programs from the benchmark and write the results. Therefore, the received input — a command with arguments — is processed and used to start the execution. In the end, writes into a file the processed results from the repair plug-in.

Evaluation module should allow through a simple command to evaluate and compare the results generated. The evaluation should take into account pertinent metrics related to the results. The comparison should provide several visual representations of the results, demonstrating the following: (1) categorization of programs fixes by type of security fault, (2) tools overlapping fixes, (3) each tool's coarse-grained performance accuracy, and (4) each tool's fine-grained performance accuracy.

Benchmark plug-in should abstract operations performed over the vulnerable programs. The ab-

stract operations map to concrete operations required by the tools. The plug-in should at least allow for the following: (1) check out a specific program to a given location, (2) compile the vulnerable program's source code, (3) provide information on the vulnerable program to be given as input to repair tools — i.e., scripts for testing, the number of tests, location of specific files.

4.2.2 Specifications

As much as we'll like to make the system generalizable to cover most of the tools, techniques, and programming languages, that's not realistic and, we should limit our approach to a specific set of requirements — i.e., strategy, validation method, errors class, programming language, and repair scope. Thus, we determine the following:

Strategy is generate-and-validate, as the repair process is quite popular for APR. Additionally, semantics-driven technology is studied less extensively, and some factors influence its effectiveness [3].

Validation Method is test-based, as is the most common method based on the organization in [2, 26] and the studies in section 2.1.8. Also, we consider this validation method for gathering a broader spectrum of potential candidate repair tools and benchmarks.

Errors class is dynamic, as these errors occur during run-time and their root cause is related to the program's semantic and behavior, such as in security faults.

Programming Languages are C/C++. Based on fig. 3.3, both C/C++ have more CWEs with uncommon issues causing undefined behavior than Java. Since C++ is an extension of the C programming language, these are considered as a group for program repair tools [57]. As well, many datasets with security faults/patches pair them together [7, 58–60].

Repair Scope must target a single vulnerable source code file to promote a fair evaluation. Therefore, the programs under repair to be evaluated must contain a vulnerability confined to a single file. The tools under evaluation do not necessarily have to perform FL, and if possible, these can assume true fault localization, by supplying the exact location of the faulty lines.

4.3 Framework

The framework is the orchestrating component of the system, responsible to handle the repair workflow for each tool, and evaluate the generated results. This section describes the considered repair tools, the implementation details, the repair workflow, and the evaluation methodology. The implementation was named SecureThemAll and is publicly available on GitHub. To access it use the following Uniform Resource Locator (URL): `https://github.com/epicosy/SecureThemAll`. Figure 4.2 illustrates the design

of the framework with the paramount components for the repair workflow.

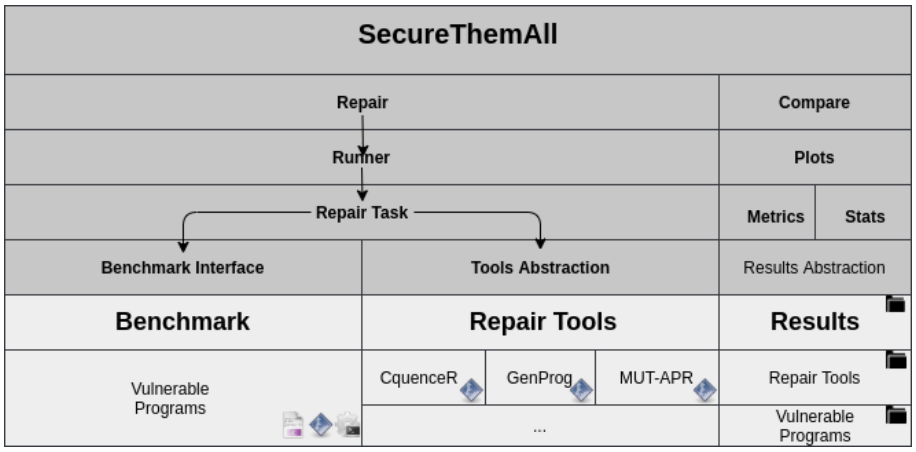


Figure 4.2: SecureThemAll Architecture (*cf.* [4]).

4.3.1 Subject Repair Tools

We leverage the organization and selection done in [3, 57], and regarding the specifications in section 4.2.2, the repair tools in Table 4.1 are selected as candidates for the framework.

Table 4.1: Generate-and-validate test-suite-based program repair tools for C/C++

Repair tool	Benchmark used in evaluation	Bugs	Patched	Fixed
GenProg [29]	ManyBugs § / SIR †	69 * / 54 †	16 * / 17 †	1 * / 17 †
MUT-APR [61] †	SIR [62]	54	47	17
Kali [63] *	ManyBugs §	69	25	2
Prophet [47]	ManyBugs §	69	39	18
RSRepair [64] ‡	ManyBugs §	30	10	2
SPR [65] *	ManyBugs §	69	38	16

* *Results selected from [47].* † *Results selected from [61].*
‡ *Results selected from [63].* § *ManyBugs Benchmark [66].*

4.3.1.1 Selection criteria

There are additional factors necessary to consider for including candidate repair tools into the framework, as indicated below.

1. The repair tool ought to be publicly available.
2. The repair tool ought to be possible to run.
3. The repair tool ought to require only the source code in C language and its test suite used as validation.

4. The repair tool ought to be easily adapted to run on a different benchmark beyond the one used in its original evaluation.

From a total of 6 tools, 2 meet the specified inclusion criteria. These are listed below with a brief description. Table 4.2 describes the rest of the tools along with the criteria not met.

Remark. Given the reduced number of tools, an additional tool named CquenceR was purposely developed and included. It's implementation details are described in the next chapter.

GenProg [29] is a seminal APR tool that uses genetic programming to repair faults. The approach uses FL to find faulty statements in a program and proceeds to modify these with mutation and crossover operators to create a new variant. The variants that pass the test suite are used for the next generation .

MUT-APR [61] mutates suspicious operators within a genetic programming algorithm to replace faulty ones. It uses the same FL appraoch as in GenProg. The operators are picked randomly, and each run generates a new copy of the program with one of them.

CquenceR is a data-driven repair technique that makes use of a NMT model trained — on patches of security faults — to generate multi-line fixes. The implementation is detailed in Chapter 5.

Table 4.2: Excluded repair tools.

Repair Tool	Criteria	Explanation
Kali [63]	1	the author states that the tool *"is available on request for the purpose of checking the results presented"*
Prophet [47]	4	programs have to be compiled with specific libraries
RSRepair [64]	2	coverage program supplied by the author does not work
SPR [65]	4	programs have to be compiled with specific libraries

4.3.2 Implementation Details

In this subsection, we describe the main components of the framework. The details about the expected underlying functionalities — i.e., OS interactions, logs, input parsing, multi-processing — are left aside.

Benchmark Interface allows the framework to interact with the benchmark, as described in section 4.2.1. The interface mimics the benchmark's operations and can be used to directly invoke it—e.g., preparing a program's working directory. The interface can be used to get operations in the format of strings for crafting the command arguments passed to the tools—e.g., formulating a program's compile and test commands required by the tool.

Repair Tool Abstraction provides the common arguments and bindings used during the Repair Workflow — i.e., benchmark binding, configurations, common methods, paths. This abstraction acts

as a toolbox that allows the framework to be extended with repair tools. The abstraction binds the results processing functionalities to the repair tool—methods for textual difference and stats parsing, and the template for the results. Some common parameters are crafted by queering the Benchmark Interface — e.g., the compile and test commands. These are written as scripts into the working directory and used by the repair tools. Other common parameters, such as the number of positive/negative tests and the file path of the file under repair, are passed directly to the tools. Some tools require specific parameters that need some workarounds — e.g., MUT-APR requires the files with the tests' outcome.

Repair Runner is responsible for the multiprocessing and receives parsed inputs along with the instantiated abstractions. It initializes the execution of the specified repair tool on a given benchmark program. The runner saves the results when the repair tool terminates its execution.

4.3.3 Repair Workflow

The overall repair workflow prepares the working directory along with the arguments passed to the repair tool. The repair tool's plugin crafts the command by using the help of the Benchmark Interface. The tool's plugin invokes a single command that starts the execution of the repair tool. The framework awaits for the tool's execution to finish. In case the time limit is passed, the framework forces the execution to terminate. When the execution ends, the results, if any, are parsed into a standard format.

Preparation and Tool Execution starts by copying into a given directory the specified program's source code. Depending on the tool, it's plugin queries from the benchmark, additional information about the program, and its test suite. Next, the plugin formulates the benchmark's compile and test commands with the necessary arguments, written to bash files, and passed as arguments for the repair tool. The respective plugin of tools executes them by simply invoking them with the crafted command.

Results Processing assesses through the repair tool's plugin the generated result after its execution terminates. The results are the generated source code files containing changes for a given vulnerable source code file. The results are compared with the original file to produce the textual differences in the format of unified diffs. In the working directory, a folder repair should contain a file, the patch for the vulnerability. We consider the rest of the files generated as edits because these didn't patch the vulnerability. Further, the repair tool abstraction processes the compile and test data generated by the benchmark as statistics. The repair runner saves the processed results into a JSON file, similar to the example in fig. A.1.

4.3.4 Evaluation Methodology

Our evaluation methodology must allow identifying the strengths and weaknesses of repair techniques to assess their ability to fix security faults. To this end, the family of criteria — for evaluating repair techniques in a comprehensive manner — introduced in [67], have been taken into consideration to build an evaluation methodology. We base the criteria for our evaluation with performance metrics calculated with statistics from the results. Framework's abstractions deal with the transformation of the generated results in statistics and metrics. We describe and justify the considered statistics and metrics below.

4.3.4.1 Statistics

The statistics make use of the results to calculate the following: *compile success rate, positive and negative tests success rate, edits score, fix score, and time score*. The scores are developed to have a range between 0 and 1.

$$compile\_success\_rate = \frac{\#successful\_compilations}{\#successful\_compilations + \#failed\_compilations} \tag{4.1}$$

$$positive\_tests\_success\_rate = \frac{\#passed\_positive\_tests}{\#passed\_positive\_tests + \#failed\_positive\_tests} \tag{4.2}$$

$$negative\_tests\_success\_rate = \frac{\#passed\_negative\_tests}{\#passed\_negative\_tests + \#failed\_positive\_tests} \tag{4.3}$$

We use *success rates* to assess the tool's robustness in generating useful code changes. For example, if the code doesn't compile, then the tool's score is penalized. We believe these give as well an insight into the quality of the patches generated by the tools.

As for the *scores*, we use the *fix* and *edits* scores to assess the tool's ability to generate useful code changes. For example, when a tool generates plenty of edits, it means it might just explore a search space. We believe that reflects the tool's ability to "reason" over the changes it performs. The *time score* serves as the time cost spent in the repair process. The *time and edits* scores are inversely proportional, which means we penalize the tools that take longer to execute and the tools that generate a great number of implausible candidate patches.

$$fix\_score = \begin{cases} 1, & \text{if } fix \in patches \\ 0, & \text{if } fix \notin patches \end{cases} \tag{4.4}$$

$$edits\_score = 10^{-\frac{edits}{50}} \tag{4.5}$$

$$time\_score = \begin{cases} 0, & \text{if } duration \leq 1 \\ 0, & \text{if } duration \geq time\_limit \wedge \varnothing patches \\ 10^{-\frac{duration}{time\_limit}}, & \text{otherwise} \end{cases} \tag{4.6}$$

4.3.4.2 Metrics

The performance metrics used to measure the ability of the repair tools are the efficiency and effectiveness. The former is calculated with eq. (4.5) and eq. (4.6), as we believe these reflect the useful work performed by a repair tool. The latter is calculated with eq. (4.4), eq. (4.6), eq. (4.1), eq. (4.2), and eq. (4.3), as we believe these reflect a tool's capacity to perform the task of repairing a vulnerable program.

$$efficiency = edits\_score + time\_score \tag{4.7}$$

$$effectiveness = compile\_success\_rate + \frac{pos\_tests\_success\_rate + neg\_tests\_success\_rate}{2} + fix\_score \tag{4.8}$$

4.3.4.3 Visual Representation

We approach the visual representations described in section 4.2.1 with plots that make use of the previous metrics and statistics. These have been matched as following.

1. **Heat map**: for representing the application fixes performed by tools according to a specific category based, by using the *fix score*
2. **Venn diagram**: for representing the number of applications repaired by tools and how these overlap
3. **Bar chart**: ranking and comparing each tool's performance according to the *efficiency* and *effectiveness* metrics.
4. **Radar chart**: represents each tool's relative performance with respect to the statistics.

4.4 Benchmark

The benchmark serves as the ground truth for evaluating the repair tools' ability to patch security faults and make a fair comparison between them. This section describes the considered benchmark, its implementation details, and the execution workflow. The implementation was named cb-repair and is publicly available on GitHub. To access it use the following URL: `https://github.com/epicosy/cb-repair`. Figure 4.3 illustrates the design of the benchmark with its paramount components.

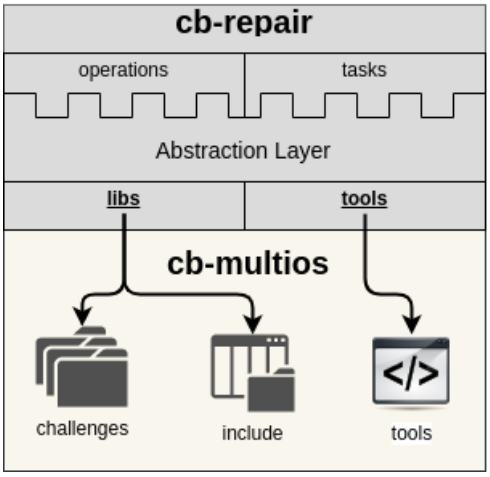


Figure 4.3: Architecture of cb-repair (*cf.* [7]).

4.4.1 Subject Benchmark

We selected from [7, 58, 68] the benchmarks listed in the table below as candidates, as these contain vulnerable programs written in C, along with the source code.

Table 4.3: Available benchmarks with C/C++ vulnerable programs.

Benchmark	Description	Programs	Defects
Codeflaws [69]	defects in programs are classified across 40 defect classes	7436	3902
DBGBench [70]	291 plausible/correct patches from real software professionals	2	27
IntroClass [66]	small programs drawn from an introductory C programming class	6	998
ManyBugs [66]	well-established open-source programs with mature codebases	9	185
CGC Corpus [7]	programs designed to contain vulnerabilities that represent a wide variety of crashing software flaws	250	739 †
C/C++ Juliet Test Suite [58]	contains examples organized under 118 different CWEs	64099 *	-

* *Test cases, which are simple programs.* † *The number of proof of vulnerabilities.*

4.4.1.1 Selection criteria

Additionally, the benchmark must contain labeled vulnerable programs with enough complexity to approximate real software, having the source code limited to a single version, to provide specific insight into security aspects of automatic repair techniques. Given the context of the problem, the creation of one is arduous work, and even adapting existing work is a tough task. We consider the factors enumerated below to cover the process of selecting a benchmark.

1. The benchmark must contain programs with security faults.
2. The benchmark must contain programs with a single version as source of data.
3. The benchmark must include, for each program, test cases to test its functionality.
4. The benchmark must include, for each program, at least one exploit that triggers the vulnerability.

Taking into account the scope of the benchmarks in table 4.3, these have a different set of concerns with respect to the evaluation.

Codeflaws is designed to *"allow thorough investigation of the relationship between fault types and the effectiveness of repair tools"* [69]. However, Codeflaws' software defects are classified based on the syntactic differences between the buggy program and the patched program.

DBGBench is designed to *"to facilitate the effective evaluation of automated repair techniques"* [70]. However, DBGBench contains programs with multiple versions of source data.

ManyBugs is designed to *"allow indicative evaluations whose results generalize to industrial bug-fixing practice"* [66].

IntroClass is designed to allow *"evaluations that can identify the factors that affect the success of repair techniques"* [66].

Juliet Test Suite is designed *"specifically for assessing the capabilities of static analysis tools"*, and the programs *are intentionally the simplest form of the flaw being tested* [58].

CGC Corpus is designed to allow the evaluation of vulnerability remediation systems.

Codeflaws, DBGBench, ManyBugs, and IntroClass contain security faults but are not dedicated entirely to them. This first impression reveals that the available benchmarks are relatively far from the desired goal. However, to a certain extent, one of them must be the closest to our scope, being only a matter of extending with an appropriate design. The CGC Corpus implemented for Linux by Trail of Bits [71] portrays that. The initial corpus was purposely built and designed for security, and it's applications come with extensive functionality tests, triggers for the security faults, and patches.

4.4.2 CGC Corpus Applications

The original DARPA Challenges Sets contains 250 C/C++ applications initially designed for DARPA's Cyber Grand Challenge (CGC) section 2.2.1. The applications approximate real software with enough complexity and represent a wide variety of crashing software flaws. The verification of the applications consists of two parts, functionality verification and vulnerability demonstration [7]. The former is made up of specialized tests—service polls—intended to check complex interactions, thus, allowing to validate the impact of the changes made to the application. The vulnerabilities are proved through special inputs purposely crafted—Proof of Vulnerability (POV)—to trigger the memory corruption in the applications. That's achieved through two types of POVs, type 1 and 2.

Type 1 POV makes the application fault, allowing to gain code execution—e.g., instruction pointer and general purpose register set to specific value
Type 2 POV leaks privileged data—e.g., bytes from a range of memory

4.4.3 Implementation Details

The initial applications were originally developed for a custom Linux-derived operating system. Trail of Bits—CGC contestants—modified the application to support other operating systems. Their work fits better within the context of program analysis and fault localization. However, the supplemental resources provided enable benchmarking of automatic repair tools at the cost of additional modifications. For that, part of the available applications and scripts have been selected and restructured. We then implement complementary extensions and organized them into an abstraction layer with specific commands, as illustrated in figure fig. 4.3.

Abstraction Layer provides commands, logic, data objects, and state modules necessary for interacting with the applications. The data objects are bind to the applications, dependencies, and scripts to allow easy accessibility. We extend the abstraction layer with concrete operations and tasks. The Operations allow the necessary interactions with the applications—e.g., test a given application. The Tasks allow to deal with specific case scenarios — e.g., initialize tests. The main difference between operations and tasks is the application's context — i.e., operations make part of an execution workflow.

Operations are designed to work on a single applications at a time, within an execution workflow and with the order below.

1. **genpolls** operation generates deterministic positive test cases.
2. **checkout** operation copies the application's source code along with other relevant assets—*manifest information, compile configurations, dependencies*—to a given working directory.
3. **compile** operation compiles the application's source code along with the negative tests. This operation allows to compile preprocessed files and link the generated objects into the application's binary—in the cases where such is necessary.
4. **info** operation provides information such as the location of compiled files, the description of the challenge and the number of positive and negative tests
5. **test** operation runs single/multiple, positive/negative tests on a compiled challenge.

Remark. For simplicity, we omit optional operations — *manifest, patch, make, test-coverage* — which apply to specific scenarios.

Tasks have a broader range, in the sense of performing specific/complex functions for multiple challenges, and might involve other tasks and operations.

catalog task lists the available applications.
check task executes the workflow aforementioned as a sanity check of applications that might fail under specific circumstances—i.e., configurations or environment.
clean task cleans metadata generate during benchmark initialization and sanity check.
init_polls task performs, for multiple applications, the same as the genpolls operation.
stats task gives and plots statistics about the benchmark's applications—e.g., distribution of the number of code lines per application

Test-based approach

The functional verification and vulnerability demonstration described in section 4.4.2 easily map to the expected positive and negative tests described in section 2.1.5. Thus, the *service polls*—here, simply called polls—are considered as **positive tests**. The underlying mechanism of polls can even execute thousands of simple and basic tests per poll, to prove the program's correct functionality and behavior.

The POVs are considered as **negative tests**. The underlying mechanism of type 1 POVs make the application terminate via a *SIGSEGV, SIGILL* or *SIGBUS* signal. Type 2 POVs demonstrated the ability to read the contents of arbitrary memory locations within the applications.

The coarse granularity of polls only turns the validation more extensive and rigorous, leaving a smaller margin for the repair tools to escape by changing original functionality. The applications come represented as state machines to guarantee that the generation of polls is deterministic and unique. That's possible by connecting the application's components into a directed graph, which allows exploring different combination. The polls are generated in XML format. The POVs come as source code in C and must be compiled to be executed. The CGC Corpus supplies the necessary tools for generating the polls, compiling the POVs, and testing both types of artifacts.

The *test* operation makes use of the supplied tooling and extends them to enable accessibility. It identifies the tests with a simple notation—e.g., *p1* identifies the first poll and *n1* identifies the first POV. The extension allows among others to execute only the positive or negative tests, to print and write specific outcomes from the execution, manipulate the execution return code, and generates stats with relevant information about tests' execution.

4.4.4 Execution Workflow

Given a working directory and a specific application from the benchmark, the simplest execution flow consists of using the following commands: *checkout*, *compile*, and *test*. That copies the application's source code with necessary assets to the working directory, compiles the application's source code with its POVs, and tests the application with a given test that exists in the test-suite. The polls are expected to be generated during the initialization of the benchmark, otherwise, these can easily be generated with the operation *genpolls*. The *info* operation is expected to be used by the repair tools after a given application is compiled—to query the number of tests or the prefix of the compiled files' location.

5

Proposed Tool for Patching Security Faults

Contents

This chapter presents an adaptation of a data-driven approach to patch security faults, which we name CquenceR. In section 5.1, we represent the problem and introduce sequence-to-sequence as an approach to patch security faults. Then in section 5.2, we describe CquenceR's implementation details. Finally, section 5.3 presents the evaluation methodology along with the dataset construction and analysis, and model's testing results.

5.1 Approach

Given our limitations with the current repair tools, our goal is to adapt a data-driven repair approach to patch security faults in C programs. As described in section 2.3, data-driven repair approaches became a trend recently, and many techniques intersect AI techniques with software engineering. Data-driven technologies are promising as these are language agnostic and can be adapted easily from one context to another. That's possible because the data and its representation weigh more rather than the model used to learn. M. Allamanis et al. [44] suggest the naturalness hypothesis, which states *"similar statistical properties shared between software corpora and natural language corpora, can be exploited to build better software engineering tools"*. That's apparent in the literature from section 2.3.1, where NMT-based approaches are used under a different context to tackle the problem of automatic software patching. We base our intuition on the fact that ML and NLP technologies have the capacity of learning patterns from a large corpus. In this section, we elaborate on that intuition, first by representing the problem, and next by supporting it with some related work, and lastly, we give a brief description of the technique considered.

5.1.1 Problem Representation

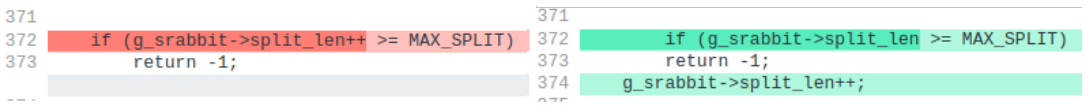


Figure 5.1: Example of a patch for CWE-190: Integer Overflow or Wraparound. On the left-side highlighted in red is the vulnerability. On the right-side highlighted in green are the corrective changes from the patch. (code snippets from program Square_Rabbit [7])

As seen in fig. 3.1, we consider that the problem of patching security faults can be outwardly represented as the following: given a vulnerable program P_v and assuming that the security issue S_i is identified by a set of ordered lines S_v = $\{l_1, l_2, ..., l_i\}$, then these can be replaced with a contiguous sequence of code lines S_p = $\{l_1, l_2, ..., l_i\}$, namely a patch, that will fix S_i in P_v such that it turns in a correctly functioning program P_c. That's represented in fig. 5.1, at line 372 on the left side highlighted with red, in the *if*

statement, the increment of the variable *split_len* can allow an integer overflow. However, the patch on the right side highlighted in green separates the variable incrementation from the check.

The problem of patching security faults at first glance seems effortless. That's a misapprehension resulting from a simplistic representation of the problem. If we go in-depth, a variety of aspects arise. For example, the granularity we considered is a contiguous snippet of lines, but overall that's not the case. Security faults come in many shapes and sizes, dispersed across multiple files, giving more shades of complexity to the problem—as is evident in section 3.1.1. As a starting point, we purposely put the blinders on to look in a constrained way at the vast context of the problem. Thus, we consider the granularity of security faults to cover a single file, spawning across multiple hunks of code — contiguous lines of code.

The changes introduced to P_v by a patch S_p will not guarantee P_c. For example, S_p might remove some functionality from P_v or change its control flow to an incorrect location. We verify these introduced changes with a test-based validation method. Lastly, the fault design we are looking for is general. We leave out the data-driven technique to capture the variety of security fault types and their peculiarities from the dataset.

5.1.2 Using Sequence-to-Sequence Learning to Patch C Security Faults

In our view, the previous representation of the problem, along with the example in fig. 5.1, share similarities with machine translation. Thus, we approach the problem of patching security faults with sequence-to-sequence learning, a branch from NMT, for several reasons. First, these are end-to-end probabilistic models easy to train with datasets of pre-and post-correction, making them easy to apply to new datasets [11]. Second, the technique has reached the necessary maturity level to deal with sequences whose characteristics significantly differ from natural language [9]. Third, are powerful models able to learn relations on a variety of granularities—e.g., token-by-token matches, to soft paraphrases [11]. Lastly, as outlined in the section 2.3, these have demonstrated success in a number of code patching problems, and even feasibility and generality for fixing software vulnerabilities [23]. These insights give us the motivation of applying sequence-to-sequence learning to the problem of patching security faults. Thus, we give a brief explanation of the sequence-to-sequence learning as a core concept and leave out the details for a more curious reader.

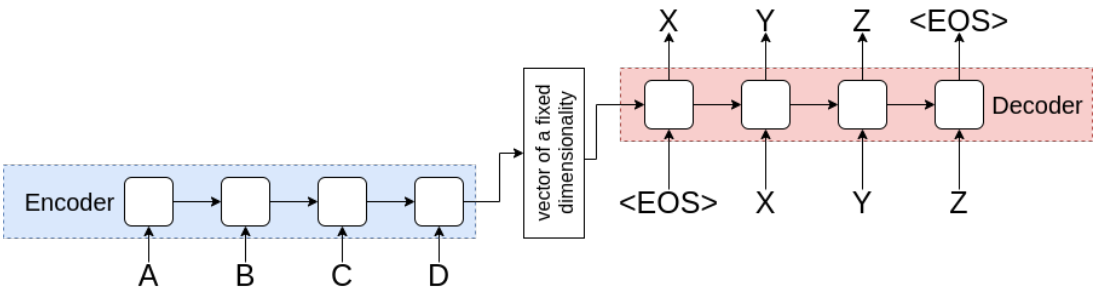


Figure 5.2: Example of early neural-based sequence-to-sequence model (*cf.* [8]).

Sequence-to-sequence learning considers input as a sequence of elements, usually words or characters, and predicts sequences, hence the name sequence-to-sequence. The sequence-based models predict sequences by sequentially generating each element [44]. Neural models are appropriate for that given their predictive performance, with the trade-off of large amounts of data for the training. The most common sequence-to-sequence paradigm with neural models consists of an encoder and a decoder. The encoder generates out of the sequence of input data a mid-level output, which is passed to the decoder to produce the final outputs [46]. Figure 5.2 illustrates an example of sequence-to-sequence based on Long Short-Term Memory (LSTM) architecture — a recurrent artificial neural network architecture.

5.2 CquenceR

We employ the intuition in the previous section by leveraging work in the literature and adapt it into a generate-and-validate tool for patching security faults. The implementation consists of two parts, learning, and repair. We train over a large corpus of security patches for vulnerabilities to learn a probabilistic model. Then, we use the model to perform translations from a sequence of faulty statements to a sequence of corrective statements. The validation follows the test-based method described in section 2.1.5. The implementation was named CquenceR and is publicly available on GitHub. To access it use the following URL: `https://github.com/epicosy/CquenceR`. Figure 5.3 illustrates the design of CquenceR along with its workflow.

5.2.1 Sequence-to-sequence model

Out of the 5 NMT-based approaches introduced in section 2.3.1, we select the state-of-the-art sequence-to-sequence model used in [9]. The model receives code in the form of a sequence of tokens as input and generates an output sequence of tokens. The model learns by capturing common patterns from the input sequence and uses the generated output sequence to predict the most likely next token given the input. The whole output sequence is generated starting from an initial token, by feeding the produced tokens in the network, as the next predicted token has a certain amount of dependency on preceding

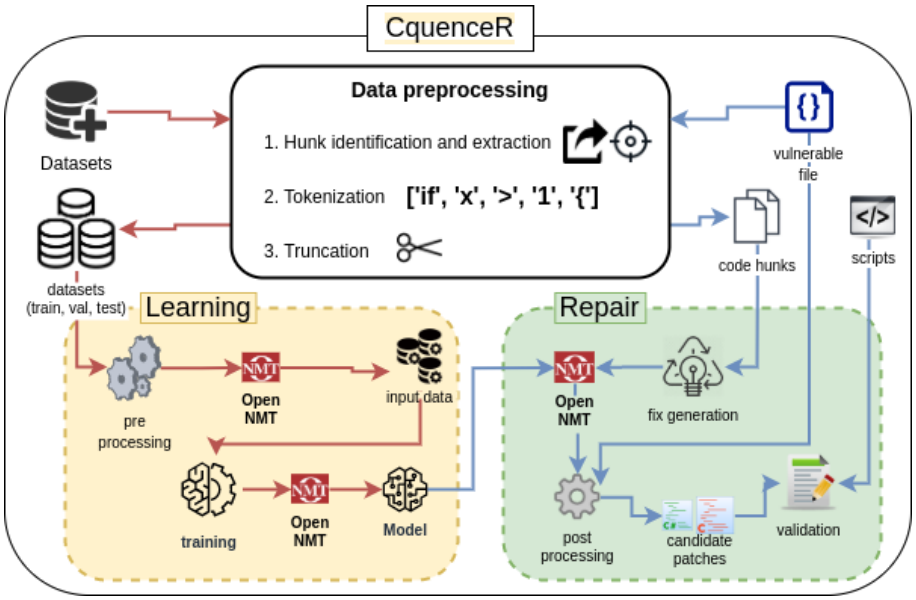


Figure 5.3: CquenceR architecture and workflow (*cf.* [8–11]).

tokens.

Our model selection takes into consideration the following: the solution must have its code publicly available and share enough similarities with our problem representation. SEQUENCER stood out, as its implementation is available on the open-science repository on GitHub along the data and scripts used for evaluation. Additionally, it deals with limitations from previous works using similar network architecture [11].

We replicate SEQUENCER's model training as documented by the authors, achieving similar results fig. A.2. As acknowledgment to the original work, we name our approach CquenceR. The main differences between SEQUENCER and CquenceR are related to the type of data and its representation, and the sizes of input/output. The original work focuses on fixing general faults, while we target vulnerabilities. The programming language we choose is C instead of Java. The initial work applies one line corrections as substitutions, while we consider substitution, addition, and removal of several continuous lines — e.g., deleting by translating multiple faulty lines to a single newline character, and vice-versa.

Further in [23], SEQUENCER's authors use sequence-to-sequence to patch security faults in vulnerable functions written in C. Although seeking the same goal, CquenceR uses the original copy mechanism for the rare word problem, while prior work uses byte-pair encoding. In CquenceR we represent the problem as a mapping between hunks of code while prior work considers different sizes of functions. As for model training, CquenceR's model learns from pairs of vulnerability patches, while prior work trains with pairs of general fault fixes.

Remark. The approach assumes true fault localization, as the scope of the adaptation is to train the repair capabilities.

5.2.2 Data Preprocessing

This step happens during both learning and repair phases, the only difference being the amount of code processed. As an overview, the input dataset is preprocessed into the pair subsets of source and target sequences, containing respectively the vulnerable and fix code hunks. The process consists of identifying vulnerable and corrective hunks in code and extracting the corrective hunk, then tokenize, and at last truncate.

Remark. The toolkit used in the learning step has it's own preprocessing. The data preprocessing is necessary as it transforms the input into the format for the training, and according to our problem representation.

Hunk identification and extraction involves identifying the vulnerable hunk and extracting the corrective hunk within the security patches. These come into the form of (unified) *diffs*—changes between code sequences. The changes can be additions or deletions and are identified with a leading $+$ or $-$. The relation between additions and deletions expresses the transformation of the vulnerable code into corrective code. Following [9] We uses the tags $TOKENIZER\_START\_VULN$ and $TOKENIZER\_END\_VULN$ to surround faulty statements and identify the vulnerable hunks. The corrective changes are extracted and segregated from the patch.

Tokenization is performed by splitting the source code with C programming language operators. This step is performed because the NMT network requires the input sequences to be in the form of separate tokens. Before the code tokenization, possible comments are removed by applying a simple regular expression. The special token $< NEW\_LINE >$ is used to represent the newline character, and to embed multiple lines into a contiguous sequence of tokens to tackle the multi-line challenge in [9].

Truncation limits the context of the input sequences, to a predetermined size and covers the cases where the input sequence is too long. This step mimics the truncation described in [9]. The rationale behind that is: code elements have dependencies; thus, including part of the original context and not only the vulnerable hunks. The choice for the truncation size limit is described in the evaluation and considers that most samples from the dataset don't require it.

5.2.3 Learning

The learning consists of iterating over the preprocessed dataset pairs and train the network. Following the original work, we implement the network with the PyTorch version of the OpenNMT project [10]. The OpenNMT toolkit gives the necessary abstraction over NMT techniques, allowing to build and train models over the command line. We leverage that to build the necessary commands for training our model. Two configurable commands are used to perform the learning, *preprocess* and *train*.

These have been implemented over OpenNMT toolkit's *onmt_preprocess* and *onmt_train* commands.

Network Parameter Settings

The following are the parameter settings for the sequence-to-sequence model [9]:

- Token embedding (uses the same embedding for both learnable recurrence function): 1,004x256 (1,000 + 4 special tokens);
- Encoder bidirectional LSTM (part of learnable recurrence function): 256x256x4x2x2;
- Decoder LSTM (part of learnable recurrence function): 512x256x4x2 + 256x256x4x2;
- Token generator bidirectional LSTM (part of learnable function): 256x1004;
- Bridge between encoder and decoder (path of first encoder hidden state to initialize first decoder hidden state): 256x256x2;
- Global Attention (attention vector weights): 256x256 + 512x256;
- Copy selector (part of learnable function): 256x1;

5.2.4 Repair

Given a source file with multiple vulnerable hunks — identified with start and end line numbers, as we assume true fault localization — the repair process consists of applying the data processing aforementioned. We identify vulnerable hunks by surrounding each with the special tokens, tokenizes the source code, and truncates each up to the defined size. The data processing output is passed as input to the learned sequence-to-sequence model to predict a specified amount of corrective sequences. These are post-processed and used to replace the vulnerable hunks in the source code and produce candidate patches — i.e., the initial file has the vulnerable hunks replaced with corrective changes. Lastly, the candidate patches are validated with the given compile and test scripts. The testing starts with negative tests, and if these passes, then positive tests are executed. When all tests pass for a candidate patch, a fix is found. The fix generation and post-processing steps are described below in more detail.

Fix Generation passes as input to the trained model the pre-processed file's output — i.e, identified and tokenized vulnerable hunks with a context up to a predefined number of tokens — to predict the corrective changes. If vulnerable hunks are too close from each other, their context overlaps; that's not a problem, the prediction gets a greater context from the vulnerability. The prediction is performed by running OpenNMT-py translation with beam search — i.e., predicts multiple likely fixes for the same vulnerable hunk. Beam search keeps the most promising sequences up to the current decoder state [9]. On top of that, the UNK tokens in the generated fix are replaced with the source tokens that had the highest attention weight, by using the flag $replace\_unk$.

Fix Post-Processing transforms the predicted sequences of tokens from the model into code syntax as these are used as ingredients in the candidate patches that must compile. Firstly, the tokens $< NEW\_LINE >$ are replaced with the newline character to turn the predicted sequences into multi-line hunks. Secondly, the spaces between operators are removed — e.g., the spaces around the **.** operator used for accessing members of a structure are removed. Lastly, each vulnerable hunk in the source code is replaced with respective predicted fix. CquenceR achieves that by keeping a mapping between the vulnerable code and it's predicted fixes, by associating an id to both and writing them to files.

5.3 Evaluation

Our approach was trained and evaluated on a great corpus of security faults. The performance of CquenceR was evaluated with consideration to the machine learning and program repair standpoints. This section describes the gathering and preparation of the data used for training, followed by the evaluation process and it's results.

5.3.1 Data Selection

We consider the selected corpus of data as a set of vulnerability-fix pairs — unified textual differences or separate files. We base our intuition on the capacity to derive from vulnerability fixes the underlying patterns for generating corrective patches for vulnerable applications. According to that, we leverage from the literature 5 datasets. These contain in total 8,798 vulnerability-fix pairs across many languages. Table 5.1 describes each dataset along with its name, the source from where the data was collected, the study which introduced it, and a brief description of the samples.

Table 5.1: Datasets of Security Patches

Name	Source	Study	Description
Mozilla	Mozilla Foundation Security Advisories	Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study [72]	Contains patches for security reports with at least one Bugzilla report or one versioning commit
SecretPatch	Common Vulnerabilities and Exposures (CVE) list	Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS [59]	Contains 1575 C/C++ security patches processed and collected from the reference URLs from CVE list.
NVD	Common Vulnerabilities and Exposures (CVE) list	A Comparative Study of Deep Learning-Based Vulnerability Detection System [73]	Contains the vulnerable and patched version for a function from each 368 open-source C/C++
Secbench	GitHub	SECBENCH: A Database of Real Security Vulnerabilities [74]	Contains 682 real security vulnerabilities from 238 projects.
MSR20	Common Vulnerabilities and Exposures (CVE) list	A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries [60]	Dataset contains 3754 C/C++ code vulnerabilities extracted from 348 Git projects.

5.3.2 Data collection and preparation

We developed and abstraction around the selected datasets to facilitate their curation. The tool allows through the command line to collect the datasets from the source, transform the data into a predefined format, and filter it according to custom rules. These steps are briefly described below. The implementation was named PatchBundle and is publicly available on GitHub. To access it use the following URL: `https://github.com/epicosy/PatchBundle`.

Collection — downloads the dataset's raw data from a configured source;
Transformation — abstracts dataset's content into well defined records within the context of code patching, with the granularity of code hunks;
Filtering — selects based on specified properties the transformed patch records into an unified dataset.

The patch record contains the following fields: project name, commit hash, year and the number of the CVE, name and extension of the file, the code hunk, the number of addition and deletion changes the hunk contains, the name of the hunk, and the number of contiguous changes.

The datasets are curated and unified into a single one by using the PatchBundle's functionalities as described above, to meet the input format for the learning step in section 5.2.3. The transformation step discards part of the raw data if it doesn't meet the format. Then the patch records are selected by C/C++

programming language — by the *.c, .h, .cpp, .cxx* extensions — and are further filtered into patches containing contiguous hunk changes with at least one type of change. Patches that don't have hunks without both additions and deletions are filtered out. A maximum limit of 20 lines is chosen for the hunk size, which accounts for most samples. As some datasets pick the same sources, the preparation takes into consideration the existence of duplicates and drops if any is found. Figure 5.4 summarizes the data preparation process. Initially, the raw data is just transformed into the patch record format, accounting for a high number of samples. Transformation and Filtering steps discard duplicates. After the preparation, there's a big discrepancy between the initial and final number of samples.

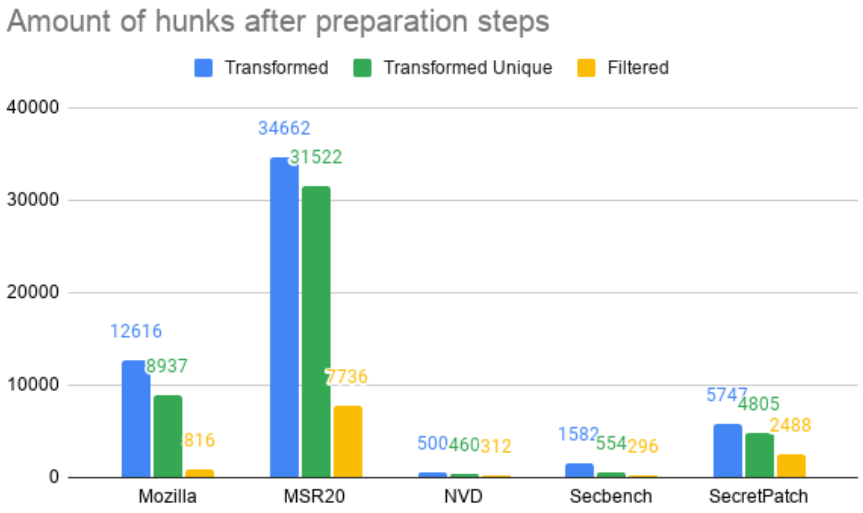


Figure 5.4: Amount of patch records in each dataset after Tranformation and Filtering steps.

5.3.3 Training Dataset Descriptive Statistics

The following statistics serve to give an insight into the data used to train the model and justify the choice of network parameters, such as the source and target sequences length, truncation limit, and vocabulary size. After the data preparation, the resulting dataset contains, in total, 7,156 samples — 18% of the total amount of examples (40,289) used in the original work. The dataset was further shuffled and split into the following subsets: 85% (6,081), 10% (716), and 5% (358) as training, validation, and testing datasets, respectively. We discard samples with empty strings during the dataset splitting, as these should at least contain the $< NEW\_LINE >$ token.

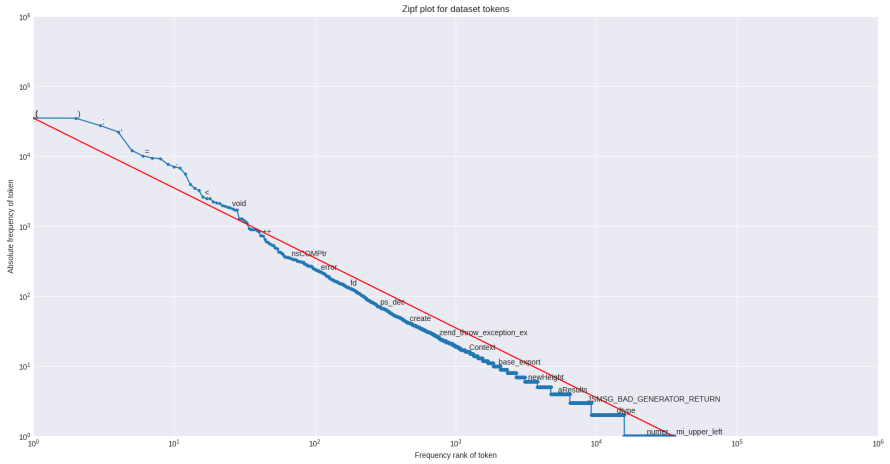


Figure 5.5: Vocabulary: tokens count occurrences

Vocabulary distribution is described in fig. 5.5 by a log-log plot with X-axis being log(rank) and Y-axis being log(frequency). The tokens occurrences follows a near Zipf's Law distribution. The full vocabulary size contains 15956 unique tokens. As observed in the figure, at the lower ranks the expected linear line crosses the 1000 rank and the observation roughly linear curve meets it. According to that the limit for the training vocabulary was set to the 1000 most common tokens, the same size as the one used by SequenceR.

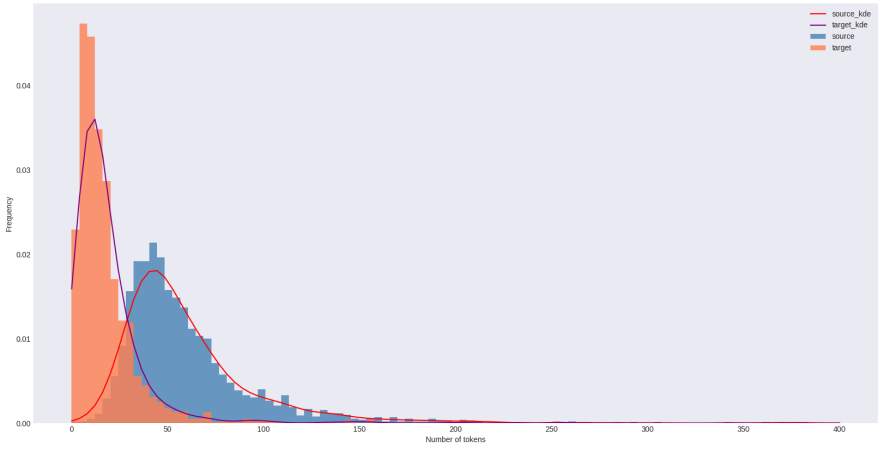


Figure 5.6: Source and target tokens count histogram

Truncation limit is based on the distribution of the number of tokens per each sample sequence in both source and target sets. As can be observed in fig. 5.6, most sequences in the source set have up

to 255 tokens and in the target set have up to 100 tokens, in percentage, that's respectively 99.68% and 99.64%.

Figure 5.7: Vulnerable hunk size in number of lines

Vulnerable hunk size is illustrated in fig. 5.7 by a bar plot with an X-axis describing the size of the vulnerable hunks in terms of numbers of statements, and Y-axis describing the number of samples. Is clear that the most vulnerable hunks are 1 line long.

5.3.4 Training

The network was trained on Google Colab with the parameters described in section 5.2.3 on a Tesla T4 GPU on the training set. We set the training to 4,000 steps, and enabled early stop with patience of 10. The total execution time was 15 minutes, and given the early stop, the execution stops at 3,050 steps. The best results for both training and validation occur at 2,350 iterations.

The accuracy and perplexity are used as metrics for evaluating how well the model performs during the training and validation. The perplexity measures how well a model predicts a sample, a low perplexity value indicates a high translation quality [9]. The accuracy and perplexity are calculated per token generated. The results from the model training are illustrated in fig. 5.8. The accuracy, for both training and validation, converges close to 80%. The perplexity for the training set converges around 2.30 and for the validation around 2.80.

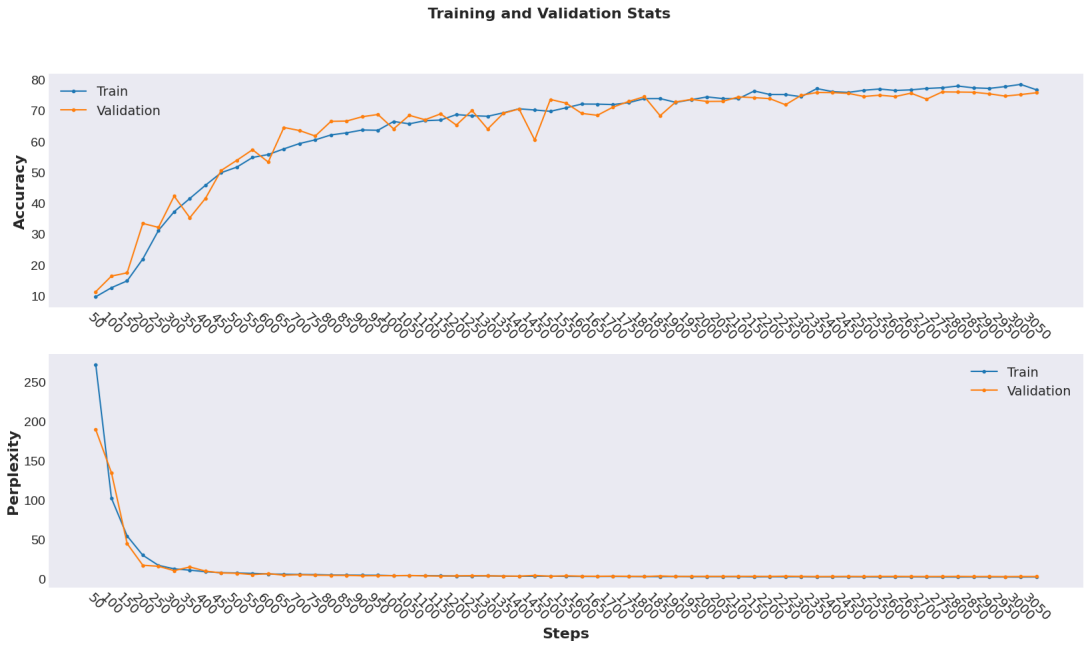


Figure 5.8: Training and validation accuracy and perplexity

5.3.5 Model Testing

The trained model was tested on the testing dataset by comparing each prediction generated with the target patch. A beam size of 50 was used for the prediction, meaning for each sample 50 predictions were made. On the testing set, the model predicted the exact patch for 95 out of 358 vulnerabilities, respectively 26.5% of the vulnerabilities. In total were predicted 17900 patches from which 218 of them were perfect patches. In the original work, the accuracy to perfectly predict the fix is 20%, with the golden configuration on a testing dataset with 4,711 samples.

Predictions similarity A similarity ratio was computed between the target patch and the predictions generated to know how effective is the approach in generating patches. The similarity is based on the Levenshtein distance [75], an edit distance that measures the difference between two strings. The Levenshtein distances considers the minimum number of operations allowed to transform one into another. The strings operations covered by Levenshtein distance are deletion, insertion and substitution — appropriate for the context. The similarity ratio for the predictions was calculated with the Python package python-Levenshtein [76] which implements the distance. The predictions generated for the testing set have a similarity average and median of 64.8% and 69.5% respectively to the target patches. In fig. 5.9 is shown the histogram for the similarity between predicted and target patches in the testing dataset, which has a peak around 85%. That demonstrates the patches generated that are not perfect fixes, still are very similar to the target patch.

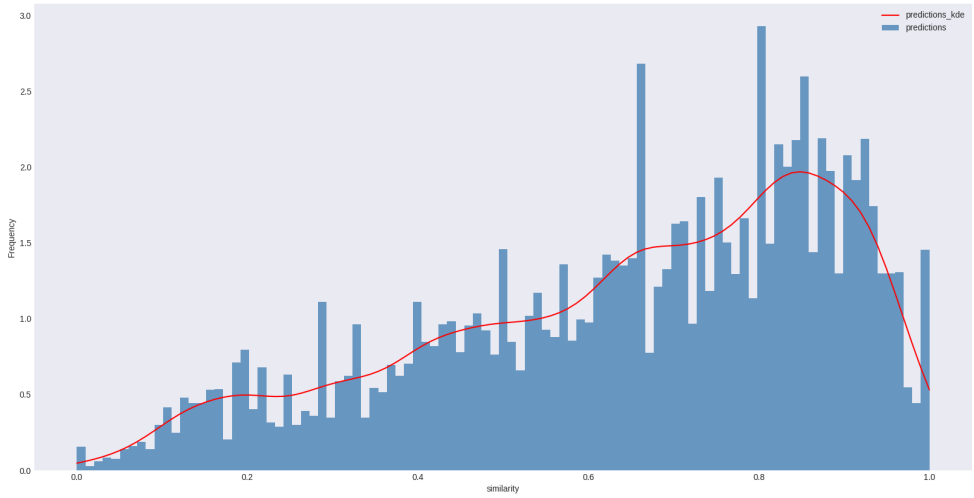


Figure 5.9: Histogram of similarity between predicted and target patches

The obtained results demonstrate our approach is able to predict perfect fixes, with a better accuracy than in the original work. That proves our intuition that underlying patterns of corrective patches can be leveraged to predict fixes for similar security faults. One threat to validity can be the existence of very similar patches within the dataset, which can bias our accuracy for predicting perfect fixes. These results are sufficient to validate our approach along other repair tools, and that is demonstrated in the following chapter.

6

Comparative Study

Contents

This chapter conducts the comparative study were we investigate the given problem for this dissertation. In section 6.1 we describe the setup used in the experimentation, along with an overview of the resources. Then in section 6.2 the results of the empirical study are reported and described with focus on the research questions. Finally, in sections 6.3 we discuss our main findings and the possible threats to their validity.

6.1 Experimental Setup

6.1.1 Benchmark

The modified CGC Corpus—by Trail of Bits [71]—contains 202 applications working on Linux. To achieve an unbiased and controlled study, the criteria—for selecting candidate applications—enumerated below have been considered.

1. **Tests Initialization**: both polls and POVs must respectively be generated and compiled without any errors;
2. **Sanity check**: tests functionality must be verified—to validate that these do what are supposed to;
3. **Single file faults**: applications' vulnerabilities must be restrained to a single file—most repair techniques fix a single file.

By applying the criteria above, 56 applications have been selected. The table below lists the number of excluded applications along a description for the unmet criteria.

Table 6.1: Excluded applications.

Number of Applications	Criteria	Explanation
52	1	generation of polls raised various errors
56	2	application's POVs were not working properly
32	3	vulnerabilities spawned across multiple files
17	2	polls testing failed during sanity check

Remark. Some of the excluded applications have more than one unmet criterion, hence the sum of numbers not adding up to the total.

Remark. Some applications don't have a state machine script for generating the polls. These have been excluded since the number of available pre-generated polls is less than what would be expected for a study.

Tests initialization generates a number of 20 positive tests per application. The benchmark was

configured to have a timeout of 60 seconds per test. The test timeout is necessary as the modifications performed by a repair tool can remove/change functionalities in the applications related to the communication with the testing tools — as these employ a networked paradigm.

Remark. During the generation of polls assertion errors related to code coverage were suppressed—a small number of applications require a high number of polls in order to cover all their code.

6.1.1.1 Descriptive Statistics

This section gives an overview of relevant information about the applications used in the study. Each application has descriptive information that includes, among others, the vulnerability's description along the CWE class associated. Some applications contain more than one vulnerability. Figure 6.1 illustrates the distribution of CWEs per challenge. The majority — i.e., 30 applications—contain one vulnerability, followed by 16 applications containing two vulnerabilities. The number of different vulnerabilities per application goes up to 6, however less occurring.

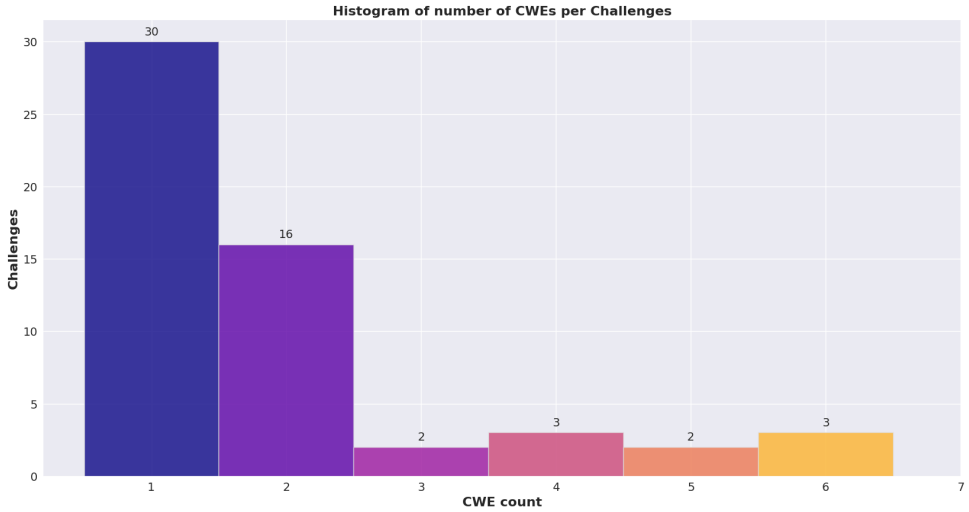


Figure 6.1: Histogram of the number of CWEs per Challenge

In total there are 34 unique CWE covering the benchmark. Those are grouped into 11 categories, as shown in fig. 6.2. That was achieved by following the hierarchy of the CWE structure. First, the Research Concepts CWE List available at [77] was parsed to capture the 'Related Weaknesses' associated with each CWE identifier. Next, each CWE identifier in the applications' description was used to recursively traverse with a depth of 3 to a more generalized type of weakness. The most present kind of weaknesses is related to CWE-664 (Improper Control of a Resource Through its Lifetime) with 36.7%, followed by CWE-118 (Incorrect Access of Indexable Resource ('Range Error')) with 23.9% and CWE-682 (Incorrect Calculation) with 14.7%. As to the top 6 most severe weaknesses in section 3.1.1, 7 applications cover

CWE-125, 5 applications cover *CWE-20*, 4 applications cover *CWE-787*, and 3 applications cover *CWE-119*. In total, the benchmark has 19 applications in the top 6 most severe weaknesses.

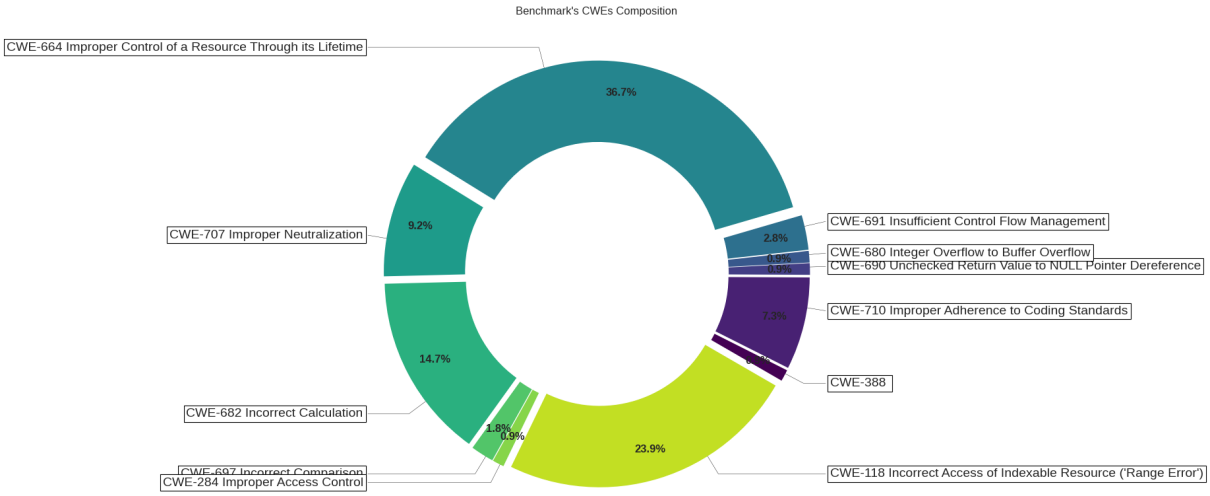


Figure 6.2: Percentage of CWEs covering the benchmark's applications

The following figures depict the distribution of the total number of code lines, the number of vulnerable lines, and patched lines across applications. These were calculated by examining the files ending with [*'.c', '.cc', '.h'*] and only the folders related to source code were searched, respectively [*'src', 'include', 'lib'*].

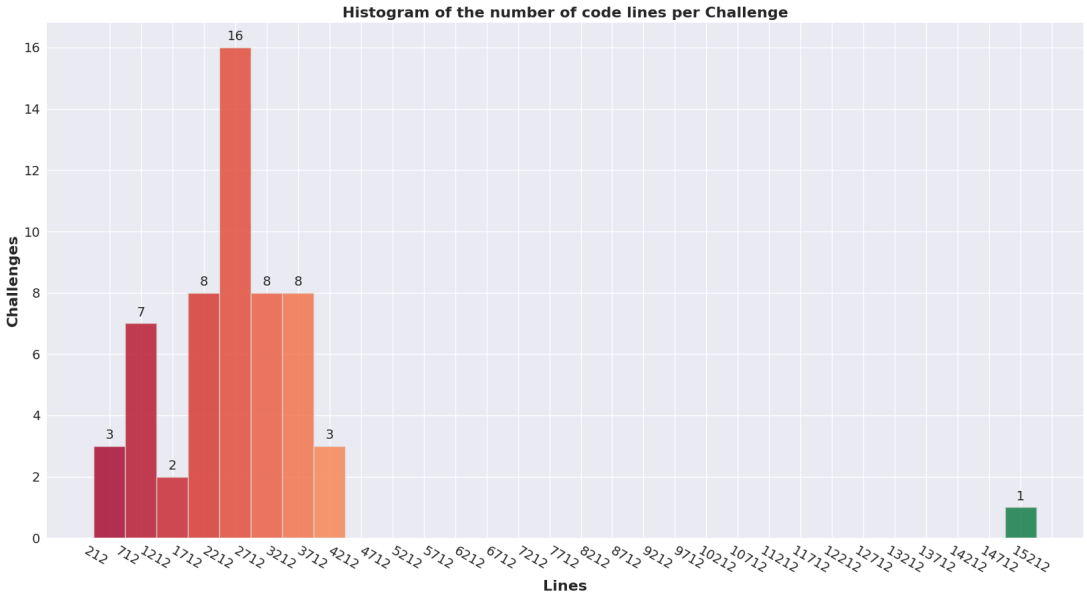


Figure 6.3: Histogram of the total number of code lines across applications

As shown in fig. 6.3, the distribution of the total number of code lines has a peak around the 2212-

2712 range. There's an outlier within the range of 14712-15212 lines of code. The vulnerable files contain enclosed lines with the ['**#ifdef, #ifndef, #elseif, #elif, #else, #endif**'] logical directives of the preprocessor. These logical operations were used for generating the vulnerable and patched compiled version of the applications. Besides that, those were leveraged to identify and count the vulnerable and patched lines of code across applications.

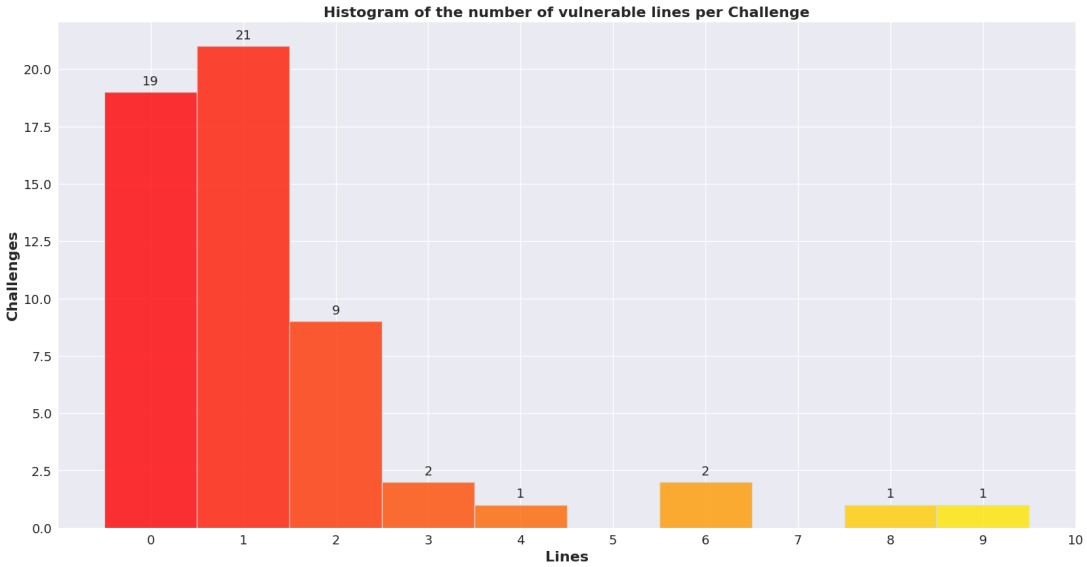


Figure 6.4: Histogram of the number of vulnerable code lines across applications

Figure 6.4 shows there are 21 applications with vulnerabilities of 1 line, followed by applications with vulnerabilities that occur because of the lack of additional code.

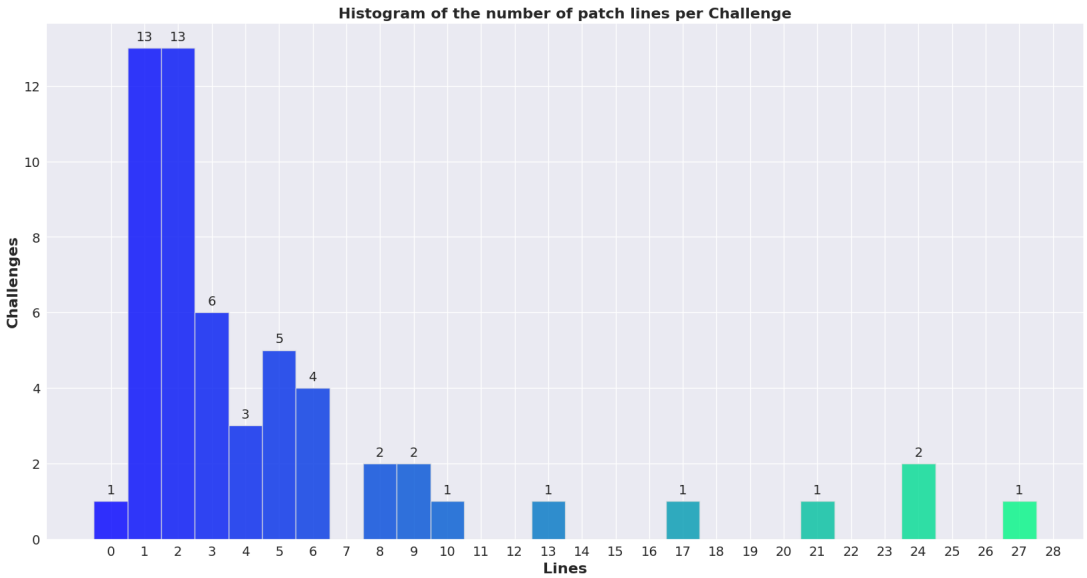


Figure 6.5: Histogram of the number of patch code lines across applications

Figure 6.5 shows the benchmark has 13 applications with a patch size of 1 line and 13 applications with a patch size of 2 lines. The application BitBlaster is the only one requiring the removal of code.

6.1.2 Framework

Within the scope of this study, we configure the framework with a timeout of 1800 seconds per tool, a timeout of 40 seconds per command, and 8 threads to use the cores of the machine to execute in parallel 8 applications. We use tool timeout to guarantee unbiased results and is necessary as tools don't have an execution timeout. We use the command timeout to avoid locks within the framework's execution. The timeout for the tool is given in eq. (6.1). We consider the worst-case scenario for the timeout, that is, a tool under analysis modifies the application making it hangs up during testing. Thus, we calculate the timeout by multiplying the number of tests with the tests' timeout, plus the estimated time sanity check takes to be performed by a tool. We calculate that by estimating a test's execution time on average, assuming a maximum (test hangs up and takes timeout) and minimum (instant, the test fails) time.

$$tool\_timeout = \#tests * test\_timeout + (\#tests * \frac{min\_time + max\_time}{2}) \quad (6.1)$$

Tools Configurations are illustrated in the figures below. The repair tools stop when they find the first fix. In fig. 6.6 the *search* is the strategy to search the program's space, the *crossover* is the operator type to create new program variants.

```
{
  "--search" : "ga",
  "--crossover" : "subset",
  "--describe-machine" : "",
  "--rep-cache": "default.cache",
  "--seed": "0",
  "--ignore-dead-code": "",
  "--no-rep-cache":"",
  "--keep-source": ""
}
```

Figure 6.6: GenProg arguments

```
{
  "--mut" : 0.01,
  "--seed" : 1
}
```

Figure 6.7: MUT-APR arguments

```
{
    "--beam_size" : "50"
}
```

Figure 6.8: CquenceR arguments

6.1.3 Environment

The system has been tested on a machine with the specifications described in the table below.

Table 6.2: Machine Specifications

Specifications	
Machine Speed	2.10 GHz.
Processor	Intel(R) Xeon(R) CPU E5-2620v4 — 8 cores used
Memory/RAM	16 GB
Disk Space	75 GB
Operating System	Ubuntu 18.04.5 LTS

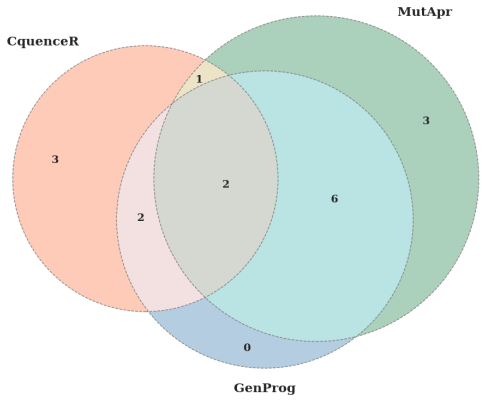
6.2 Results

This section illustrates and describes the results of the comparative study for 3 subject APR tools — MUT-APR, GenProg, and CquenceR. We conduct the study with the solution proposed in chapter 4 and following the setup described in section 6.1. The results are mapped as an answer to the research questions presented in section 4.1.1. The execution time of the overall workflow for each tool took 44 minutes for CquenceR, 1 hour and 57 minutes for MUT-APR, and 2 hours and 36 minutes for GenProg.

Remark. A repair/patch refers to a plausible fix, as we perform a quantitative study on the results generated by the tools and processed by the framework, based on the metrics proposed in section 4.3.4.2. Thus, we rely on the system as a whole to conduct this study.

6.2.1 Repairability

We demonstrate with a Venn diagram the *repairability* of state-of-art repair tools for security faults. Figure 6.9(a) illustrates the number of repairs achieved by each tool as overlapping sets. MUT-APR repairs 12 applications (21.4%), followed by GenProg with 10 repairs (17.9%), and then CquenceR with 8 repairs (14.3%). From the diagram is clear that GenProg only repairs programs that MUT-APR and CquenceR repair. Note that there are two programs repaired by all tools, two programs repaired only by CquenceR and GenProg, and one program repaired only by CquenceR and MUT-APR. Figure 6.9(b) illustrates the number of overlapped repaired applications per repair tool in comparison with the remaining tools. That is, MUT-APR repairs 80% of applications that are also repaired by GenProg, and 66.7% vice-versa. MUT-APR repairs 37.5% of applications that are also repaired by CquenceR, and 25% vice-versa. CquenceR repairs 40% of applications that are also repaired by GenProg, and 50% vice-versa.



(a) Venn diagram of patched applications per repair tool.

	CquenceR	GenProg	MutApr
CquenceR	37.5% (3)	40% (4)	25% (3)
GenProg	50% (4)	0%	66.7% (8)
MutApr	37.5% (3)	80% (8)	25% (3)

(b) Table of the overlapped patched applications per repair tool.

Figure 6.9: The number of overlapped patched challenges per repair tool

6.2.2 Specificity

The specificity of the repair tools' results is demonstrated with the heatmap in section 4.3.4.3, to answer the question *"What categories of security faults are the state-of-art repair tools able to patch?"*. The applications are categorized by common CWE classes and grouped into a heatmap. The columns in the heatmaps represent the applications under repair, and the rows represent the subject repair tools. The heat map score can assume two values, 1 for fix and 0 otherwise.

Figure 6.10 groups a considerable part of applications under the common *CWE-664*, which accounts for 36.7% of the vulnerabilities in the benchmark. In this category, MUT-APR repairs 6 out of 24 applications, which is 25%. GenProg repairs 4 applications, which is 16.7%, and CquenceR repairs 2 applications, which is 8.3%.

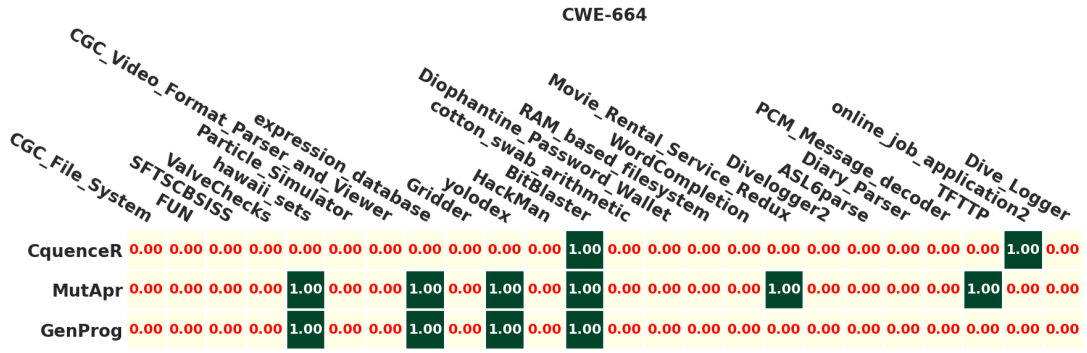


Figure 6.10: Tools' repair of applications with CWE-664: Improper Control of a Resource Through its Lifetime

Given the tools' low fix percentages, we relaxed the correction criteria and considered corrections

that correct the flaws with some compromise in the program's functionality. In fig. 6.11, the vertical axis represents the number of corrections and the horizontal axis represents the percentage of positive tests that pass, taking into account that all negative tests pass. We also plotted the continuous probability density curve for each tool's results in the figure, for a better understanding of the results. The curves in fig. 6.11 for GenProg and CquenceR have a denser distribution between the range of 0% to 30%, which is, their patches greatly affect a program's functionality. In contrast, MUT-APR's curve has a peak around 100%.

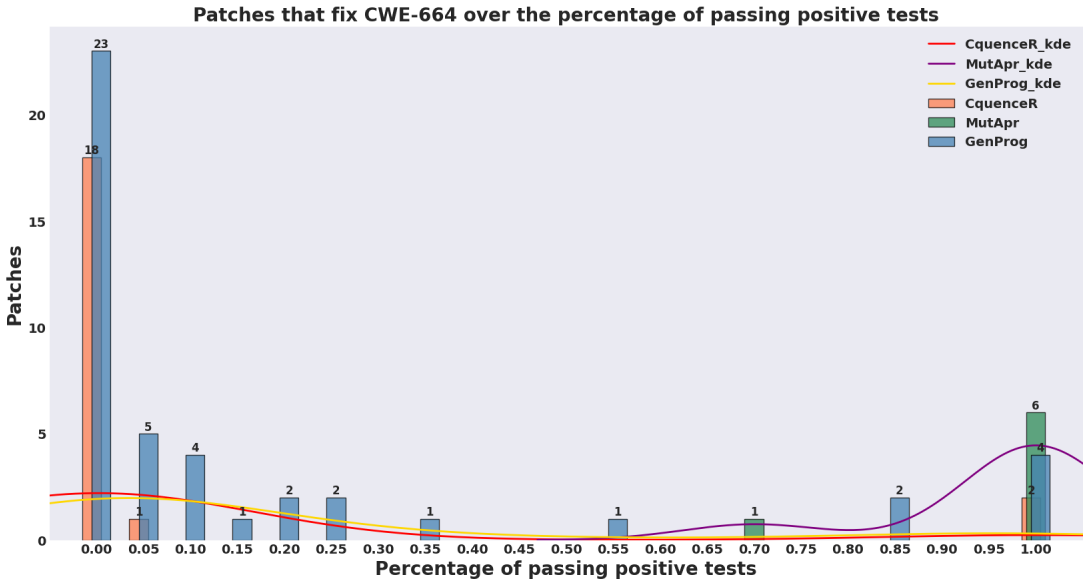


Figure 6.11: Patches that fix CWE-664 over the percentage of passing positive tests.

The table in fig. 6.12 explains fig. 6.11 in numbers. GenProg generates 45 patches, and 48.9% of them maintain 5% or more of a program's functionality, and 15.6% of the patches maintain 50% or more of a program's functionality. MUT-APR has a higher success rate, 100% of the patches generated by the tool fixes the CWE-664 fault and maintains 50% or more of a program's functionality, and 85.7% of those patches maintain 100% of a program's functionality. CquenceR generates 21 patches and 14.3% of them maintain 5% or more of a program's functionality.

	Total patches fixing faults	Patches fixing faults that maintain program's functionality		
		>= 5 %	>= 50%	Fixes (100%)
CquenceR	21	3 (14.3%)	2 (9.5%)	2 (9.5%)
GenProg	45	22 (48.9%)	7 (15.6%)	4 (8.9%)
MutApr	7	7 (100%)	7 (100%)	6 (85.7%)

Figure 6.12: Table with the amount of patches fixing CWE-664 faults per tool.

Figure 6.13 illustrates the other considerable group of applications under the common *CWE-118*, which accounts for 23.9% of the vulnerabilities in the benchmark. In this category, GenProg repairs 4 out of 17 applications, which is 23.5%. CquenceR repairs 3 applications, which is 17.6%, and MUT-APR repairs 2 applications, which is 11.8%.

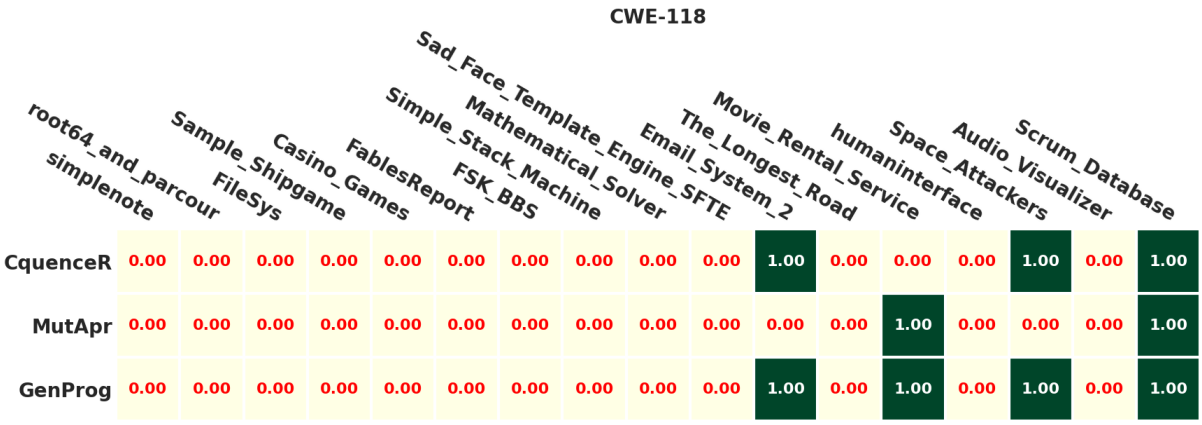


Figure 6.13: Tools' repair of applications with CWE-118: Incorrect Access of Indexable Resource ('Range Error')

The probability distribution curves in fig. 6.14 for GenProg and CquenceR are quite smooth, and that demonstrates their patches have a fairly dispersed probability in terms of a program's affected functionality. In contrast, MUT-APR's patches affect less a program's functionality.

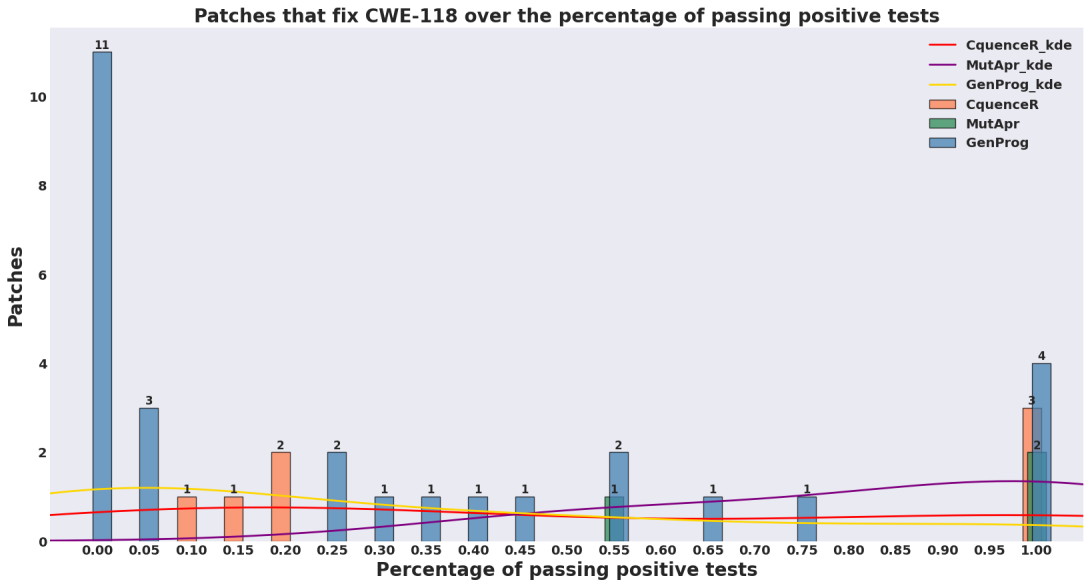


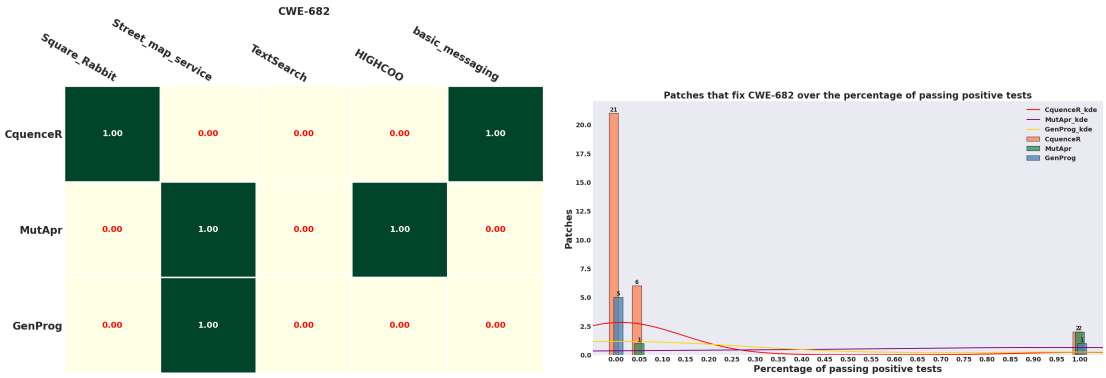
Figure 6.14: Patches that fix CWE-118 over the percentage of passing positive tests.

In the table from fig. 6.15, 100% of MUT-APR patches maintain 50% or more of a program's functionality, followed by CquenceR with 42.9%, and GenProg with 28.6%. GenProg generates more patches that fix CWE-118 fault in comparison with CquenceR and MUT-APR, and 60.7% of them maintain 5% or more of a program's functionality. In contrast, all MUT-APR and CquenceR patches maintain 5% or more of a program's functionality.

	Total patches fixing faults	Patches fixing faults that maintain program's functionality		
		>= 5 %	>= 50%	Fixes (100%)
CquenceR	7	7 (100%)	3 (42.9%)	3 (42.9%)
GenProg	28	17 (60.7%)	8 (28.6%)	4 (14.3%)
MutApr	3	3 (100%)	3 (100%)	2 (66.7%)

Figure 6.15: Table with the amount of patches fixing CWE-118 faults per tool.

The heatmap in fig. 6.16(a) groups the applications with vulnerabilities within the class *CWE-682*. For this category, both CquenceR and MUT-APR repair 2 out of 5 applications, which is 40%, and GenProg 1 application, which is 20%. Figure 6.16(b) illustrates that most CquenceR and GenProg patches that fix the fault inhibit a program's functionality. For programs with CWE-682, MUT-APR generates one patch that maintains 5% of a program's functionality.

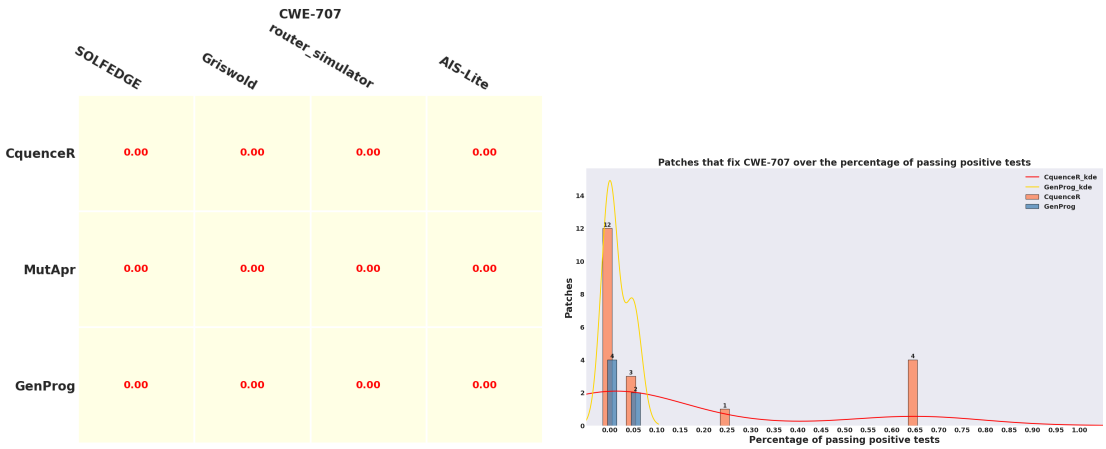


(a) Tools' repair of applications with CWE-682: Incorrect Calculation

(b) Patches that fix CWE-682 over the percentage of passing positive tests.

Figure 6.16: Heatmaps with the rest of the classes of vulnerable applications in the benchmark along with the number of patches that fix the fault but compromise a percentage of a program's functionality

The heatmap in fig. 6.17(a) groups the applications with vulnerabilities within the class *CWE-707*. For this category, no repair tool can fix the programs such that all tests pass. Figure 6.17(b) illustrates that most CquenceR and GenProg patches that fix the fault inhibit a program's functionality. CquenceR generates 4 patches that fix the fault and maintain 65% of a program's functionality. MUT-APR does not generate patches that fix CWE-707.



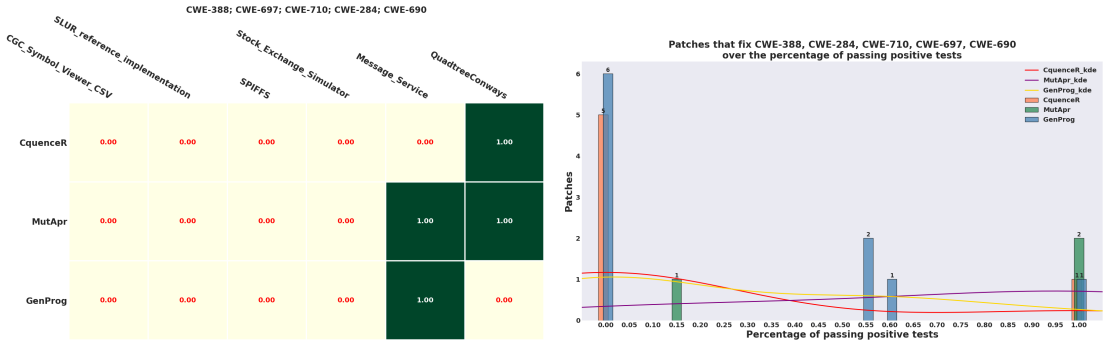
(a) Tools' repair of applications with CWE-707: Improper Neutralization

(b) Patches that fix CWE-707 over the percentage of passing positive tests.

Figure 6.17: Heatmap with the rest of the classes of vulnerable applications in the benchmark along with the number of patches that fix the fault but compromise a percentage of a program's functionality

The heatmap in fig. 6.18(a) groups the rest of the applications with vulnerabilities within the classes *CWE-697, CWE-710, CWE-388, CWE-284, CWE-690*. For these categories, MUT-APR repairs 2 out of 6 applications, which is 33.3%. Both Cquencer and GenProg fix 1 application, which is 16.7%.

Figure 6.18(b) illustrates that most CquenceR and GenProg patches inhibit a program's functionality. GenProg generates 2 patches that fix the fault and maintain 55% of a program's functionality, and 1 patch that maintains 60%. MUT-APR generates one patch that maintains 15% of a program's functionality.



(a) Tools' repair of applications with other CWE categories

(b) Patches that fix the rest of CWE classes over the percentage of passing positive tests

Figure 6.18: Heatmaps with the rest of the classes of vulnerable applications in the benchmark along with the number of patches that fix the fault but compromise a percentage of a program's functionality

6.2.3 Suitability

We approach repair tools' suitability with bar and radar charts as described in section 4.3.4.3, to illustrate which paradigm fits better for fixing security faults, with different levels of accuracy. Figure 6.19 ranks based on the metrics introduced in section 4.3.4.2, thus, the score goes up to 6, as there are 3 ratios and 3 scores that range between 0 and 1. However, the highest score is 2.3 and is achieved by GenProg, followed by MUT-APR with 2.1, and then CquenceR with a score close to 2.1. As we base the rank on two metrics, we can rank according to efficiency or effectiveness. Thus, MUT-APR is the most effective tool with a score of 1.7, followed by GenProg with 1.6, and then CquenceR with 1.2. Regarding efficiency, CquenceR is the most efficient tool with a score of 0.9, followed by GenProg with 0.7, and then MUT-APR with 0.4.

Figure 6.20 illustrates the tools' profiling based on the ratios and scores introduced in section 4.3.4.1. That gives us a better insight into the tool's performance. MUT-APR has the highest success rate for positive tests, meaning it doesn't compromise functionality. GenProg and MUT-APR have a high compile success rate. That means their change operators don't introduce new syntactic errors, in contrast with CquenceR. That seems to give an advantage on the time score for CquenceR, justifying its efficiency. As for the edit score, both MUT-APR and CquenceR have a low score comparing to GenProg, meaning it generates fewer candidate patches. In terms of negative tests success rate, CquenceR has a higher score in comparison to MUT-APR and GenProg, meaning its patches can pass more negative tests.

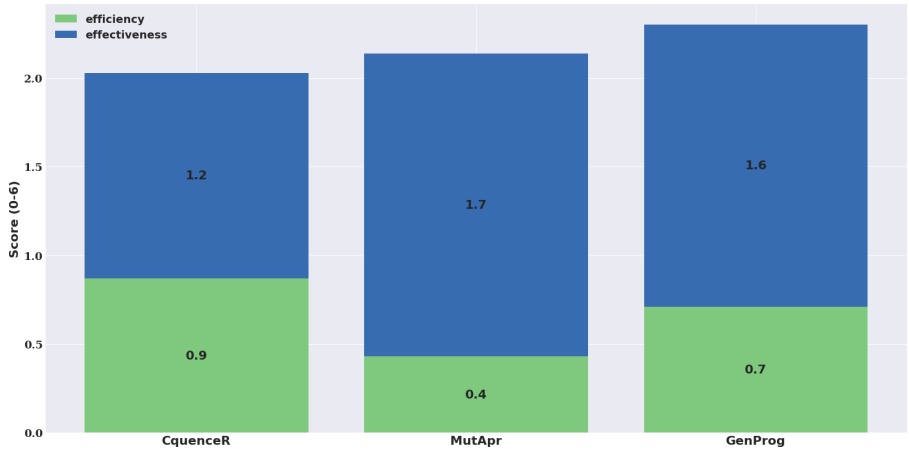


Figure 6.19: Repair tools' performance

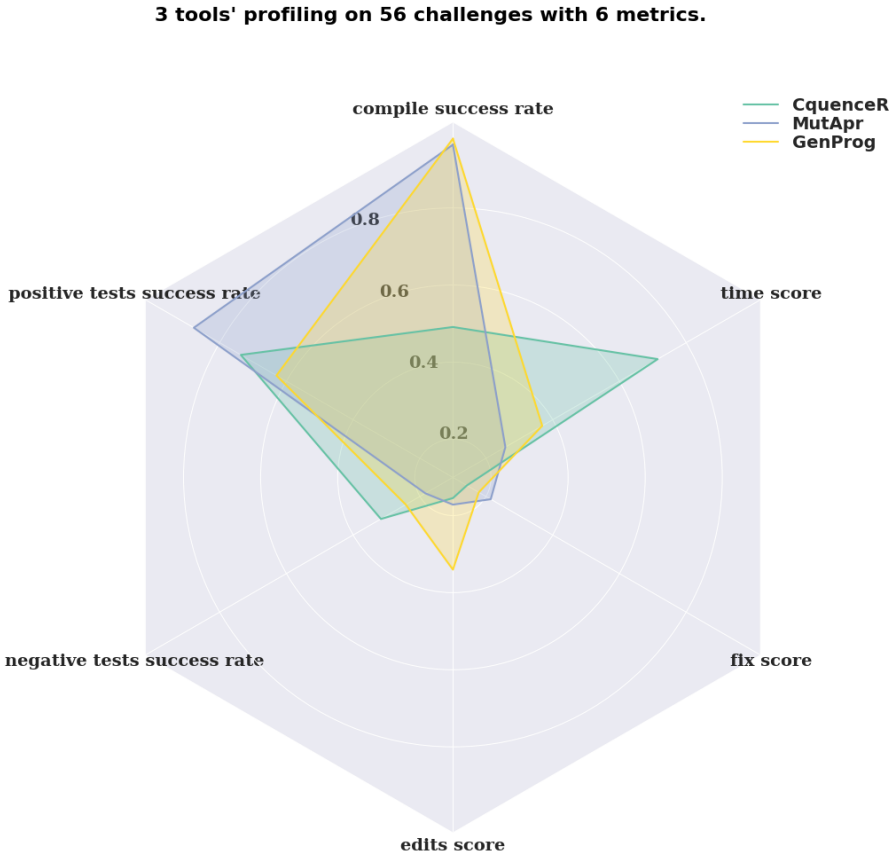


Figure 6.20: Radio chart profiling the repair tools based on assessment metrics

6.3 Discussion

In this comparative study, as an answer to **Q1**, we demonstrate that state-of-art repair tools can patch security faults to a certain extent, that is, these can fix individually at most 21.4%, and together 30.4% of applications in a benchmark containing 56 vulnerable applications, these among the top 6 2020 most dangerous software weaknesses. In this study, as an answer to **Q2**, we found the specific categories of security faults that state-of-art tools can fix, these are, *CWE-664 Improper Control of a Resource Through its Lifetime*, *CWE-118: Incorrect Access of Indexable Resource ('Range Error')*, *CWE-682 Incorrect Calculation*, and Other less common weaknesses — i.e., *CWE-690, CWE-388, CWE-697, CWE-284*. We also found that the current state-of-art techniques are not able to fix *CWE-707: Improper Neutralization* faults. Another interesting finding regarding **Q2**, is a program's affected functionality from a fault fixing patch generated by a state-of-art repair tool. We found that MUT-APR generates fewer patches that fix the fault, and overall these maintain a program's functionality. In contrast, CquenceR and GenProg, generate more patches that fix the fault, and overall most of them considerably affect a program's functionality. The state-of-art repair tools considered use a generate-and-validate strategy and a test-based validation method. In particular, the families of the underlying techniques are mutation-based, genetic programming, and data-driven. As an answer to **Q3**, we found based on the performance metrics considered, that the most suitable approach for security faults is genetic programming. However, in terms of effectiveness, data-driven perform better.

6.3.1 Threats to validity

During the conduction of this study, the availability and flexibility of repair tools played a crucial role. Although our best efforts, we were able to consider only 2 out of 6 repair tools. That required us to adopt the third tool from previous work for our study. That is a threat to validity, as we have described in Chapter 2 because the APR corpus is broad, and we obtain our discoveries from a small subset. Thus, practitioners in the field need to consider the community when developing APR tools because these allow to conduct similar studies and generate knowledge in several directions. The following are aspects that might have influence over the results reported in section 6.2:

The system might not be free of faults, as with any implementation. Several iterations of the comparative study and deliberate analysis of evident anomalies have been performed before reporting the results in section 6.2. That allowed to remove critical design/technical faults biasing the results. The system and processed results are publicly available for potential users to check the validity of our study.

The validation method we use in this study might not be sound. Test-based repair considers patches generated that make the test suite pass. Although the applications are considered as fixed

by patches that pass the test suite, that doesn't guarantee patch quality as demonstrated in previous studies [63]. The generated patches correctness is out of the scope of this dissertation. However, that doesn't invalidate the conclusions of the comparative study conducted in this chapter.

Execution duration is an important factor that determines the correctness of the applications. In this study, the applications are assessed for their activity with a timeout. We use a timeout to indicate when to terminate the applications' execution that do not respond. The timeout is generalizable over all tests, thus, if tools alter significantly the execution duration of the applications, then these might suffer penalties that will reflect in the results.

7

Conclusion

Contents

7.1 Conclusions

In this dissertation, we start by exploring security faults within the context of Automatic Program Repair. That leads us to the first descriptive case study comparing automatic software repair tools for fixing security faults.

We analyze preliminary studies and find that these did not explore enough security faults nor a common baseline standard for evaluating them. We contextualize the problem on security faults, and that reveals some uncharted territory. That raises some questions which help us break up our initial problem into attainable issues. These relate to the assessment, evaluation, and comparison of program repair tools on security faults. We see the open challenges in them and proceed to formulate our objective. That is, to develop a system that permits the assessment, evaluation, and comparison of APR tools' ability to fix programs with security faults and conduct a comparative study based on it.

We leverage representative work to conduct such task and implement the system that consists of a framework for repair tools, that we name *SecureThemAll*, and a benchmark with vulnerable programs that we name *cb-repair*. Given the difficulty of adaptation and lack of repair techniques within our context, we leverage and adopt a data-driven repair tool that we name *CquenceR* to populate our framework. We conduct the first case study with the proposed system to investigate the repairability, specificity, and suitability of 3 repair tools to fix security faults. Our study demonstrates their potential for repairing critical flaws, as the literature addresses less these faults. Our findings lead us to conclude that state-of-the-art repair tools that use a test-based validation method can repair security faults but are farther from being effective. However, developers can use these tools in debugging and testing tasks to correct security flaws, with the cost of compromising part of a programs' functionality. Also, the genetic programming technique is more effective, and the data-driven is more efficient for repairing security faults.

7.2 System Limitations and Future Work

We have confined our work to repair techniques that follow a generate-and-validate strategy. Semantic-driven techniques have a noticeable background [3] and have been previously evaluated in empirical studies [4]. Currently, our system is limited to a generate-and-validate approach as the underlying benchmark dependencies implement the negative tests as executable binaries. Semantic-driven approaches require expected input and output. Our work opens opportunities for future investigations. First, we make publicly available the solutions implemented in this dissertation and practitioners can validate our findings, and further extend our work with, for instance, approaches that we had issues table 4.2. Secondly, we propose an assessment, evaluation, and comparison methodology that can be employed in other contexts. Lastly, we encourage the use of a common language such as CWE listings to report research findings.

Bibliography

[1] "About cve," Dec 2020. [Online]. Available: https://cve.mitre.org/about/index.html

[2] M. Monperrus, "The living review on automated program repair," HAL/archives-ouvertes.fr, Tech. Rep. hal-01956501, 2018.

[3] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[4] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 302–313. [Online]. Available: https://doi.org/10.1145/3338906.3338911

[5] F. L. Loaiza, J. D. Birdwell, G. L. Kennedy, and D. Visser, "Utility of artificial intelligence and machine learning in cybersecurity," Institute for Defense Analyses, Tech. Rep., 2019. [Online]. Available: http://www.jstor.org/stable/resrep22692

[6] "Cwe list version 4.3," Dec 2020. [Online]. Available: https://cwe.mitre.org/data/index.html

[7] B. Caswell. Cyber grand challenge corpus. [Online]. Available: http://www.lungetech.com/cgc-corpus/

[8] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[9] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *ArXiv*, vol. abs/1901.01808, 2019.

[10] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proc. ACL*, 2017. [Online]. Available: https://doi.org/10.18653/v1/P17-4012

[11] H. Hata, E. Shihab, and G. Neubig, “Learning to generate corrective patches using neural machine translation,” *ArXiv*, vol. abs/1812.07170, 2018.

[12] J. Zwolak. (2000, October) Program Bug Examples. Accessed 21-October-2020. [Online]. Available: http://courses.cs.vt.edu/~cs1206/Fall00/bugs_CAS.html

[13] S. Castro. (2016, October) Off-by-one overflow explained. Accessed 21-October-2020. [Online]. Available: https://csl.com.co/en/off-by-one-explained/

[14] K. Panetta. (2020, December) 5 Trends Appear on the Gartner Hype Cycle for Emerging Technologies, 2019. Accessed 28-December-2020. [Online]. Available: https://www.gartner.com/smarterwithgartner/5-trends-appear-on-the-gartner-hype-cycle-for-emerging-technologies-2019/

[15] S. Sneha, L. Malathi, and R. Saranya, “A survey on malware propagation analysis and prevention model,” *International Journal of Computer Applications*, vol. 131, pp. 23–27, 2015.

[16] S. Karnouskos, “Stuxnet worm impact on industrial cyber-physical system security,” *IECON Proceedings (Industrial Electronics Conference)*, 11 2011.

[17] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, “How to design a program repair bot? insights from the repairnator project,” *CoRR*, vol. abs/1811.09852, 2018. [Online]. Available: http://arxiv.org/abs/1811.09852

[18] F. Gao, L. Wang, and X. Li, “Bovinspector: Automatic inspection and repair of buffer overflow vulnerabilities,” in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 786–791.

[19] M. Zhang and H. Yin, “Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications,” in *NDSS*, 2014.

[20] S. Ma, D. Lo, T. Li, and R. H. Deng, “Cdrep: Automatic repair of cryptographic misuses in android applications,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 711–722. [Online]. Available: https://doi.org/10.1145/2897845.2897896

[21] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, “Safe memory-leak fixing for c programs,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 459–470.

[22] J. Harer, O. Ozdemir, T. Lazovich, C. P. Reale, R. L. Russell, L. Y. Kim, and S. P. Chin, “Learning to repair software vulnerabilities with generative adversarial networks,” *CoRR*, vol. abs/1805.07475, 2018. [Online]. Available: http://arxiv.org/abs/1805.07475

[23] Z. Chen, S. Kommrusch, and M. Monperrus, “Using sequence-to-sequence learning for repairing c vulnerabilities,” *ArXiv*, vol. abs/1912.02015, 2019.

[24] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, “Vurle: Automatic vulnerability detection and repair by learning from examples,” in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 229–246.

[25] M. Monperrus, “Automatic software repair: a bibliography,” *CoRR*, vol. abs/1807.00515, 2018. [Online]. Available: http://arxiv.org/abs/1807.00515

[26] Y. Liu, L. Zhang, and Z. Zhang, “A survey of test based automatic program repair,” *Journal of Software*, vol. 13, pp. 437–452, 08 2018.

[27] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[28] A. Smirnov and T. Chiueh, “Automatic patch generation for buffer overflow attacks,” in *Third International Symposium on Information Assurance and Security*, 2007, pp. 165–170.

[29] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[30] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 691–701.

[31] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, “Not all bugs are the same: Understanding, characterizing, and classifying the root cause of bugs,” *ArXiv*, vol. abs/1907.11031, 2019.

[32] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, “Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset,” *CoRR*, vol. abs/1811.02429, 2018. [Online]. Available: http://arxiv.org/abs/1811.02429

[33] J. Yi, S. H. Tan, S. Mechtaev, M. Bohme, and A. Roychoudhury, “A correlation study between automated program repair and test-suite metrics,” ser. ICSE ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 24. [Online]. Available: https://doi.org/10.1145/3180155.3182517

[34] X. Kong, L. Zhang, W. E. Wong, and B. Li, “The impacts of techniques, programs and tests on automated program repair: An empirical study,” *Journal of Systems and Software*, vol. 137, pp. 480 – 496, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217301279

[35] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *CoRR*, vol. abs/1812.08693, 2018. [Online]. Available: http://arxiv.org/abs/1812.08693

[36] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, “A critical review on the evaluation of automated program repair systems,” *Journal of Systems and Software*, vol. 171, p. 110817, 2021. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121220302156

[37] B. Liu, L. Shi, Z. Cai, and M. Li, “Software vulnerability discovery techniques: A survey,” in *2012 Fourth International Conference on Multimedia Information Networking and Security*, Nov 2012, pp. 152–156.

[38] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, “V-fuzz: Vulnerability-oriented evolutionary fuzzing,” *ArXiv*, vol. abs/1901.01142, 2019.

[39] “Static exploration of taint-style vulnerabilities found by fuzzing,” in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, Aug. 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/shastry

[40] T. N. Brooks, “Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems,” *CoRR*, vol. abs/1702.06162, 2017. [Online]. Available: http://arxiv.org/abs/1702.06162

[41] DARPA. THE WORLD’S FIRST ALL-MACHINE HACKING TOURNAMENT. [Online]. Available: https://archive.darpa.mil/cybergrandchallenge/

[42] T. Avgerinos, D. Brumley, J. Davis, R. Goulden, T. Nighswander, A. Rebert, and N. Williamson, “The mayhem cyber reasoning system,” *IEEE Security Privacy*, vol. 16, no. 2, pp. 52–60, 2018.

[43] T. Ji, Y. Wu, C. Wang, X. Zhang, and Z. Wang, “The coming era of alphahacking?: A survey of automatic software vulnerability detection, exploitation and patching techniques,” in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, 2018, pp. 53–60.

[44] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” *CoRR*, vol. abs/1709.06182, 2017. [Online]. Available: http://arxiv.org/abs/1709.06182

[45] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, “Hoppity: Learning graph transformations to detect and fix bugs in programs,” in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=SJeqs6EFvB

[46] A. Torfi, R. A. Shirvani, Y. Keneshloo, N. Tavaf, and E. A. Fox, "Natural language processing advancements by deep learning: A survey," 2020.

[47] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 298–312. [Online]. Available: https://doi.org/10.1145/2837614.2837617

[48] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 213–224.

[49] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 727–739. [Online]. Available: https://doi.org/10.1145/3106237.3106253

[50] J. Devlin, J. Uesato, R. Singh, and P. Kohli, "Semantic code repair using neuro-symbolic transformation networks," *CoRR*, vol. abs/1710.11054, 2017. [Online]. Available: http://arxiv.org/abs/1710.11054

[51] Z. Yu, M. Martinez, T. F. Bissyandé, and M. Monperrus, "Learning the relation between code features and code transforms with structured prediction," *CoRR*, vol. abs/1907.09282, 2019. [Online]. Available: http://arxiv.org/abs/1907.09282

[52] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," *CoRR*, vol. abs/1607.02902, 2016. [Online]. Available: http://arxiv.org/abs/1607.02902

[53] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[54] M. Corporation. (2020, October) About CWE. Accessed 22-October-2020. [Online]. Available: https://cwe.mitre.org/about/index.html

[55] Dec 2020. [Online]. Available: https://cwe.mitre.org/data/slices/699.html

[56] Aug 2020. [Online]. Available: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

[57] S. Mechtaev. (2020, December) Program Repair Tools. Accessed 24-December-2020. [Online]. Available: https://program-repair.org/tools.html

[58] N. C. for Assured Software. (2020, December) Test Suites. Accessed 26-December-2020. [Online]. Available: https://samate.nist.gov/SARD/testsuite.php

[59] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, “Detecting ”0-day” vulnerability: An empirical study of secret security patch in oss,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 485–492.

[60] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “A c/c++ code vulnerability dataset with code changes and cve summaries,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 508–512. [Online]. Available: https://doi.org/10.1145/3379597.3387501

[61] F. Y. Assiri and J. M. Bieman, “An assessment of the quality of automated program operator repair,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 273–282.

[62] K. El-Faramawi and L. Maggiore. (2020, December) Software-artifact Infrastructure Repository. Accessed 27-December-2020. [Online]. Available: http://sir.csc.ncsu.edu/php/previewfiles.php

[63] Z. Qi, F. Long, S. Achour, and M. 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*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 24–36. [Online]. Available: https://doi.org/10.1145/2771783.2771791

[64] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, “The strength of random search on automated program repair,” ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 254–265. [Online]. Available: https://doi.org/10.1145/2568225.2568254

[65] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 166–178. [Online]. Available: https://doi.org/10.1145/2786805.2786811

[66] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, “The manybugs and introclass benchmarks for automated repair of c programs,” *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[67] Y. Qi, W. Liu, W. Zhang, and D. Yang, “How to measure the performance of automated program repair,” 07 2018, pp. 246–250.

[68] S. Mechtaev. (2020, December) Program Repair Benchmarks. Accessed 22-December-2020. [Online]. Available: https://program-repair.org/

[69] Shin Hwei Tan, Jooyong Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 180–182.

[70] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 11th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 1–11.

[71] K. El-Faramawi and L. Maggiore. (2020, December) DARPA Challenge Binaries on Linux, OS X, and Windows. Accessed 26-December-2020. [Online]. Available: https://github.com/trailofbits/cb-multios

[72] H. Alves, B. Fonseca, and N. Antunes, "Software metrics and security vulnerabilities: Dataset and exploratory study," in *2016 12th European Dependable Computing Conference (EDCC)*, 2016, pp. 37–44.

[73] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103 184–103 197, 2019.

[74] S. Reis and R. Abreu, "Secbench: A database of real security vulnerabilities," in *SecSE@ESORICS*, 2017.

[75] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet physics. Doklady*, vol. 10, pp. 707–710, 1965.

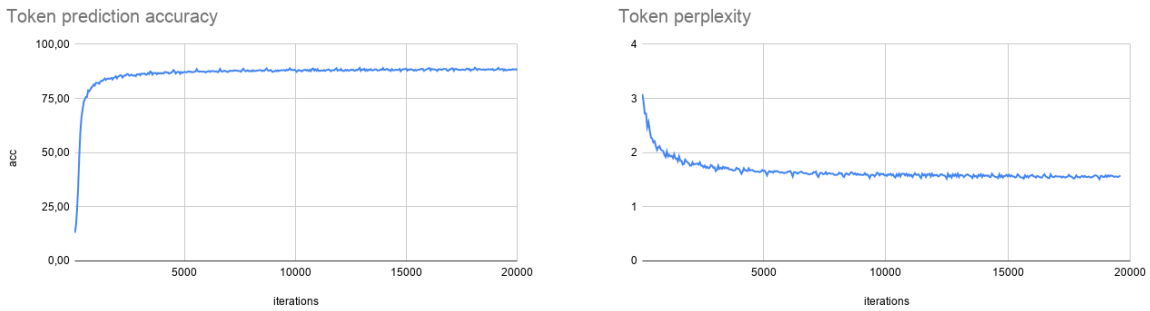
[76] A. Haapala. (2020, December) The Levenshtein Python C extension. Accessed 28-December-2020. [Online]. Available: https://github.com/ztane/python-Levenshtein

[77] M. Corporation. (2020, November) CWE Downloads. Accessed 10-November-2020. [Online]. Available: https://cwe.mitre.org/data/downloads.html

Artifacts

```
{
  "repair_begin": "2021-01-12 13:53:06.664504",
  "repair_end": "2021-01-12 13:59:48.582585",
  "patches": [
    {
      "target_file": "src/main.i",
      "fix": "346c346\n<   (*((int (*)())0))();\n---\n> \n",
      "edits": [
        "165a166\n>   {\n166a168,169\n>   gBoard[31] &= 4294967292U;\n>   }\n",
        "293a294\n>   {\n294a296,297\n> \n>   }\n",
        "309c309\n< \n---\n>       return (1);\n313c313\n<   return (1);\n---\n> \n",
        "346c346\n<   (*((int (*)())0))();\n---\n> \n"
      ]
    }
  ],
  "compilations": 7,
  "failed_compilations": 0,
  "outcomes": {
    "f8f06783": {
      "outcome": 0,
      "neg_tests": {
        "n1": 1
      },
      "pos_tests": {
        "p1": 1,
        "p2": 1,
        "p3": 1,
        "p4": 1,
        "p5": 1,
        "p6": 1,
        "p7": 1,
        "p8": 1,
        "p9": 1,
        "p10": 1
      }
    }
  },
  "neg_tests": 1,
  "pos_tests": 20,
  "passed_neg_tests": 2,
  "passed_pos_tests": 105,
  "failed_neg_tests": 5,
  "failed_pos_tests": 35,
  "duration": 401.918081,
  "errors": []
}
```


Figure A.1: Example of the processed results format (we include just the fix outcome in the example for 10 tests, as the of the complete outcomes for the example is too extensive).



(a) Training accuracy.

(b) Training perplexity.

Figure A.2: Replication results for SequenceR's model training with the golden configurations. (*cf.* [9])