UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Automatic Removal of Flaws in Embedded System Software

João Maria Martins Inácio

Mestrado em Segurança Informática

Dissertação orientada por:
Prof.ª Doutora Ibéria Vitória de Sousa Medeiros

2022

Acknowledgments

The end of this dissertation represents the culmination of an experience full of emotions, ups and downs, moments of inspiration and frustration, which contributed certainly to my personal growth.

First of all, I would like to thank my advisor, Prof.ª Ibéria Medeiros, for her guidance and feedback, valuable advice, and support throughout this work. Particularly for motivating me and always being very interested in helping me in everything in her reach so that I could finish this dissertation.

I would also like to thank my girlfriend, Rafaela, for all the support she has given me throughout these years. For supporting my decisions and always encouraging me to do my best. And especially for motivating me to finish this dissertation and contributing to my mental sanity by distracting me when I needed it most.

Last but not least, I thank my family and friends for all the support and affection they have always given me and for understanding my absences at some periods. I thank my parents, who always tried to provide me with the best education possible. Without them, I would never have gotten this far. Thank you for everything.

This work was partially supported by the national funds through P2020 with reference to the ITEA3 European through the XIVT project (I3C4-17039/FEDER- 039238), and through FCT with reference to LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020).

To all those who contributed, directly or indirectly, to the realization of this work.

Resumo

O avanço das tecnologias de informação e o crescimento da utilização de sistemas de *software* a nível mundial levantam várias questões relacionadas com a segurança do *software* utilizado. O uso diário de dispositivos de *software* tem aumentado significativamente ao longo dos anos. Na nossa vida quotidiana, recorremos a vários dispositivos cujo funcionamento assenta e depende do *software*, tais como *smartwatches*, *smartphones*, computadores, entre outros. Todos estes dispositivos estão em constante desenvolvimento e evolução, procurando sempre proporcionar novas funcionalidades e uma melhor experiência de utilização aos seus utilizadores. O *software* tem vindo a tornar-se mais robusto e complexo para fornecer estas novas características. No entanto, o aumento da complexidade e do tamanho do seu código favorecem o aparecimento de *bugs*, uma vez que este se torna mais difícil de analisar e de assegurar de que está correto. Sob certas condições a que os sistemas são submetidos, estes *bugs* podem causar o aparecimento de vulnerabilidades exploráveis que conduzem à corrupção do sistema. A exploração de uma vulnerabilidade pode ter efeitos catastróficos, dependendo do tipo de sistema que a contém. Se for um sistema de segurança crítico, como o de um automóvel autónomo, a sua corrupção pode causar a perda de um valor monetário considerável para os fabricantes, ou mesmo pior, conduzir à perda de vidas humanas.

A existência de *bugs* nos sistemas ocorre devido à utilização de linguagens de programação inseguras e à inserção de erros não intencionais pelos programadores. Muitos destes erros foram cometidos no passado, quando a segurança não era um conceito tão importante e onde, ainda, não existia muita literatura sobre princípios de programação segura. Embora hoje em dia exista esta preocupação com a segurança do *software*, as linguagens de programação inseguras ainda são amplamente utilizadas e erros continuam a ser cometidos, os quais continuam a ser um dos principais problemas na construção de sistemas seguros.

Uma vez que os sistemas são construídos pelos humanos e cometer erros faz parte da natureza humana, existirão sempre erros, por muito pequenos que sejam. O aparecimento destes erros pode estar relacionado com o facto de os programadores terem um tempo limitado para realizar os projetos de *software* que lhes são atribuídos ou com falta de conhecimento/informação sobre conceitos de segurança e programação segura. Como o desenvolvimento dos projetos de *software* têm limitações de tempo e recursos, por vezes não existe a possibilidade de testar o *software* adequadamente, deixando alguns *bugs* no código, os quais podem estar na base de possíveis vulnerabilidades que, se encontradas por atacantes, podem ser exploradas. Algumas linguagens de programação contêm funções que podem ser utilizadas para evitar a introdução de vulnerabilidades e, assim, invalidar ataques. No entanto, os programadores podem não conhecer tais funções, nem saber como as utilizar corretamente.

Atualmente, existe uma grande procura de ferramentas que ajudem a desenvolver *software* seguro

para ultrapassar as dificuldades acima mencionadas. No entanto, tais ferramentas são usualmente difíceis de utilizar e reportam vulnerabilidades que não são reais, ou seja, falsos positivos. Por esta razão, muitas destas ferramentas exigem que os programadores analisem manualmente os seus resultados, o que lhes consome uma quantidade significativa de tempo, onde este muitas vezes é gasto a verificar vulnerabilidades inexistentes no código.

A existência de ferramentas capazes de detetar e corrigir automaticamente vulnerabilidades facilitaria as tarefas dos programadores e diminuiria o tempo necessário para escrever código seguro. No entanto, existem poucas ferramentas disponíveis com estas capacidades, e as que existem têm algumas limitações, nomeadamente a produção de código sintaticamente incorreto ou a não verificação da eficácia das correções geradas.

A linguagem de programação C é uma das mais utilizadas para o desenvolvimento de *software* de produtos de diversas áreas, tais como sistemas operativos, controladores de *hardware*, e sistemas incorporados. Mesmo com o aparecimento de novas linguagens, o C continua a ser uma das mais utilizadas. Simultaneamente, a linguagem C carece de mecanismos de proteção, deixando toda a responsabilidade da gestão correta da memória e dos recursos para o programador. Devido a estes aspetos, existem muitas vulnerabilidades nos programas desenvolvidos em C. De acordo com o "*Top 25 of the most dangerous weaknesses*"criado pela *Common Weakness Enumeration* (CWE), as vulnerabilidades mais próximas do topo relacionadas com a linguagem C dizem respeito a *buffer overflows*. Por conseguinte, é necessário encontrar formas de corrigir estas vulnerabilidades, removendo-as do código, para tornar o código desenvolvido com esta linguagem de programação mais seguro. Além disso, é também necessário, por um lado, confirmar a existência das vulnerabilidades encontradas para reduzir os falsos positivos e, por outro lado, verificar se as correções geradas estão corretas e se são eficazes na remoção das vulnerabilidades.

O principal objetivo desta dissertação é criar uma ferramenta capaz de detetar automaticamente as vulnerabilidades de *buffer overflow* em programas escritos em C, corrigindo as vulnerabilidades encontradas, e verificando a eficácia das correções geradas. Para desenvolver esta ferramenta, dividimos o objetivo principal em três sub-objetivos: O primeiro consiste no estudo das vulnerabilidades de *buffer overflow* na linguagem de programação C, as diferentes funções disponíveis nesta linguagem que são consideradas inseguras contra este tipo de vulnerabilidades, as versões destas funções tipicamente consideradas seguras e como utilizá-las corretamente, e como criar novas formas de proteção. O segundo consiste no estudo de técnicas de deteção de vulnerabilidades no código fonte, gerando casos de teste para confirmar tais vulnerabilidades e o código que será inserido para as remover. O último consiste na conceção e implementação da ferramenta desejada e na realização de uma avaliação experimental do protótipo desenvolvido para analisar o seu desempenho.

A solução proposta combina técnicas de análise estática com *fuzzing* para identificar vulnerabilidades com maior precisão. A solução analisa o código fonte de um programa através de uma ferramenta de análise estática (*Flawfinder*), para procurar potenciais vulnerabilidades relacionadas com *buffer overflows*. A partir dos resultados desta análise são gerados pequenos programas que contêm as potenciais vulnerabilidades encontradas. Estes programas são exercitados com um *fuzzer* (AFL), que gera casos de teste destes pequenos programas e os executa nestes, visando filtrar as potenciais vulnerabilidades e verificar quais são exploráveis (ou seja, vulnerabilidades reais), providenciando os seus *exploits* (os casos de teste que as exploraram). De seguida, os programas das vulnerabilidades reais são processados

e analisados estaticamente, para identificar os locais no código onde elas existem e recolher informações necessárias para gerar as respetivas correções. Através desta informação, as correções são geradas e aplicadas nos respetivos programas. Posteriormente, os programas são novamente exercitados com o *fuzzer*, utilizando os *exploits*, que anteriormente exploravam as vulnerabilidades existentes nestes programas, em conjunto com outros novos gerados a partir de mutações destes, com a finalidade de verificar se as correções foram geradas e inseridas corretamente e se são eficazes, ou seja, se estão sintaticamente corretas e removem as vulnerabilidades com sucesso. Finalmente, quando termina todo este processo de validação, as correções sintaticamente corretas e eficazes são inseridas no código do programa recebido inicialmente, resultando numa versão nova e corrigida do programa original.

Para avaliar o protótipo desenvolvido, utilizámos um conjunto de 1075 pequenos programas escritos em C, recolhidos do *Software Assurance Reference Dataset* (SARD), para medir o desempenho da ferramenta e validar as suas capacidades. Para além disso, utilizámos seis aplicações reais, retiradas do site *SourceForge*, e ainda um *driver* de um subsistema de controlo de propulsão ferroviário, disponibilizado por um parceiro do projeto XIVT, do qual faz parte este trabalho. Estas aplicações permitiram analisar e avaliar o comportamento da ferramenta com código real. Os resultados experimentais mostram que a ferramenta foi capaz de detetar vulnerabilidades relacionadas com *buffer overflow* e corrigi-las eficazmente. Foram identificadas 6 vulnerabilidades de dia-zero nas seis aplicações reais. Todas as correções aplicadas pela ferramenta foram geradas corretamente, o código gerado estava sintaticamente correto, e as vulnerabilidades existentes foram removidas com sucesso.

Com base nestes resultados, concluímos que a nossa solução satisfaz os objetivos propostos para este trabalho e pode ser uma mais-valia para trabalhos relacionados com a correção automática de código. Além disso, pode ser uma ferramenta útil para melhorar a qualidade do código e a segurança de *software* que venha a ser desenvolvido no futuro.

Palavras-chave: Vulnerabilidades de *Buffer Overflow*, Análise Estática, *Fuzzing*, Correcção de Código, Segurança de *Software*

Abstract

Currently, embedded systems are present in a myriad of devices, such as Internet of Things, drones, and Cyber-physical Systems. The security of these devices can be critical, depending on the context they are integrated and the role they play (e.g., water plant, car). C is the core language used to develop the software for these devices and is known for missing the bounds of its data types, which leads to vulnerabilities such as buffer overflows. These vulnerabilities, when exploited, cause severe damage and can put human life in danger. Therefore, the software of these devices must be secure.

One of the concerns with vulnerable C programs is to correct the code automatically, employing secure code that can remove the existing vulnerabilities and avoid attacks. However, such task faces some challenges after finding the vulnerabilities, namely determining what code is needed to remove them and where to insert that code, maintaining the correct behavior of the application after applying the code correction, and verifying that the generated code correction is secure and effectively removes the vulnerabilities. Another challenge is to accomplish all these elements automatically.

This work aims to study diverse types of buffer overflow vulnerabilities in the C programming language, forms to build secure code for invalidating such vulnerabilities, including functions from the C language that can be used to remove flaws. Based on this knowledge, we propose an approach that automatically, after discovering and confirming potential vulnerabilities of an application, applies code correction to fix the vulnerable code of those vulnerabilities verified and validate the new code with fuzzing/attack injection.

We implemented our approach and evaluated it with a set of test cases and with real applications. The experimental results showed that the tool detected the intended vulnerabilities and generated corrections capable of removing the vulnerabilities found.

Keywords: Buffer Overflow Vulnerabilities, Static Analysis, Fuzzing, Code Correction, Software Security

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Acronyms

ACC	Accuracy
AFL	American Fuzzy Lop
API	Application Programming Interface
ASLR	Address Space Layout Randomization
AST	Abstract Syntax Tree
CWE	Common Weakness Enumeration
DEP	Data Execution Prevention
FN	False Negatives
FNR	False Negative Rate
FP	False Positives
FPR	False Positive Rate
LIFO	Last In, First Out
PCS	Propulsion Control Subsystem
PFP	Possible False Positive
Pr	Precision
RFP	Real False Positive
SARD	Software Assurance Reference Dataset
SEH	Structured Exception Handler
SEHOP	Structured Exception Handler Overwrite Protection
TN	True Negatives
TNR	True Negative Rate
TP	True Positives
TPR	True Positive Rate

Chapter 1

Introduction

The advancement of technologies and the growth in the use of software systems globally raises several questions related to the security of the software used. The usage of software devices daily has grown significantly over the years. In our everyday life, we use several devices whose operation depends on the software they use, such as smartwatches, smartphones, computers, and others. All these devices are in constant development and evolution, always searching for bringing new features and a better user experience. The software becomes more robust and complex to provide these new features. The increase in complexity and size favors the appearance of bugs in code since it becomes harder to analyze and ensure that it is correct. Under certain conditions to which systems are submitted, these bugs can cause the appearance of exploitable vulnerabilities leading to the corruption of the system. The exploitation of a system can have catastrophic effects depending on its type. If it is a critical safety system like an autonomous car, its corruption can cause the loss of large amounts of money for manufacturers, or even worse, lead to the loss of human lives.

The existence of bugs in systems occurs due to the usage of insecure languages and unintentional errors introduced by programmers. Many of these mistakes were made in the past when security was not such an important concept. There was not much literature on safe programming principles yet. Although today there is this concern with software security, unsafe programming languages are still widely used, and errors continue to be made and are one of the main problems in building secure systems.

Since systems are made by humans and making mistakes is part of human nature, there will always be errors, however small they may be. The appearance of these errors could be due to the limited time that developers have to carry out the projects or the lack of knowledge or defective information about concepts of security and secure programming. As projects have time and resource limitations, sometimes there is no possibility to test the software properly, leaving some bugs to be found and possible vulnerabilities to attackers exploit. Some programming languages contain functions that can be used to remove vulnerabilities and invalidate attacks. However, some developers have no knowledge of these functions or do not even know how to use them.

Currently, there is a great demand for tools that help the development of secure software to overcome the difficulties mentioned above. However, such tools can be hard to use and can report vulnerabilities that are not real, i.e., false positives. For this reason, many tools require developers to manually analyze the reported results, which consumes a significant amount of developers' time. Moreover, this time is ineffective when they look for inexistent vulnerabilities in the source code.

The existence of tools capable of automatically detecting and fixing vulnerabilities would make developers' tasks easier and decrease the time needed to write secure code. However, there are few tools available with these capabilities [38][51], and those that exist have some limitations, such as producing syntactically incorrect code or not verifying that the generated fixes are effective [28][50].

1.1 Motivation

The use of software daily has become inevitable nowadays. Almost every tool we use in our daily lives depends on software, whatever the area, such as medicine or telecommunications. The C programming language is one of the most used languages for developing software for products in diverse areas, such as operating systems, drivers, and embedded systems. Even with the appearance of new languages, it remains one of the most used, as stated in Tiobe Index [15].

At the same time, C lacks protection mechanisms, leaving the entire responsibility to the developer for the correct management of memory and resources. Because of these aspects, there are many vulnerabilities in programs developed in C. According to the "Top 25 of the most dangerous weaknesses" from the Common Weakness Enumeration (CWE), the C language vulnerabilities closest to the top relate to buffer overflows [6]. Therefore, it is necessary to find appropriate ways to remove these vulnerabilities by correcting the code and thus making the code developed with this programming language more secure. Furthermore, it is also necessary, on the one hand, to confirm the existence of the vulnerabilities found for the reduction of false positives and, on the other hand, to verify the correctness and effectiveness of the corrections made.

This dissertation focuses on studying buffer overflow vulnerabilities in the C language, the functions available in this language considered insecure against these vulnerabilities and its deemed secure versions, and the ways of writing secure code for invalidating those vulnerabilities. Moreover, it focuses on the development of a tool capable of automatically detecting and confirming buffer overflow vulnerabilities in C programs and removing them by correcting the code of the programs and checking such corrections. Hopefully, helping to make the software more secure.

1.2 Objectives

The main objective of this dissertation is to create a tool capable of automatically detecting buffer overflow vulnerabilities in C programs, correcting the vulnerabilities found, and verifying the effectiveness of the corrections. To develop such a tool, we can divide this main objective into three sub-objectives:

- The first objective is to study buffer overflow vulnerabilities in the C language, the different functions available in this language considered insecure against these vulnerabilities, the versions of these functions considered secure and how to use them correctly, and how to create new forms of protection.

- The second objective is to study techniques for searching for vulnerabilities in source code, generating test cases to confirm such vulnerabilities and the code that will be inserted to remove them.

- The last objective is to design and implement the desired tool based on the information gathered in the previously mentioned studies and to conduct an experimental evaluation of the developed prototype to analyze its performance.

1.3 Contributions

The main contributions of this dissertation are the following:

- A study on buffer overflow vulnerabilities in the C programming language, the different functions available in this language considered insecure against these vulnerabilities, the versions of these functions considered secure, and the situations where they should or should not be used.
- An approach for searching for potential buffer overflow vulnerabilities based on static analysis and their confirmation based on the generation of test cases derived from fuzzing. In addition, the approach includes the automatic creation of fixes to remove the vulnerabilities, their application, and assessment of their effectiveness.
- A tool capable of detecting and confirming buffer overflow vulnerabilities in programs written in C, correcting the vulnerabilities found, and verifying the effectiveness of the corrections in an automated way.
- An experimental evaluation of the tool, using synthetic test cases and real applications, to assess its performance.

This research led to a fast abstract, for now, entitled *Effectiveness on C Flaws Checking and Removal*, in the 52nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'22) [34].

1.4 Document Structure

This document has the following structure:

- Chapter 2 analyzes some related work and main concepts relevant to the focus of this dissertation, namely a study on vulnerabilities, vulnerability detection, and automatic software repair.
- Chapter 3 explains our proposed solution to detect and correct vulnerabilities, the challenges it faces, and how we address them. It also describes the solution's architecture and its main components. In addition, it presents the way components interact among them.
- Chapter 4 presents our solution's implementation in detail.
- Chapter 5 illustrates the evaluation and validation of the implemented prototype. It also discusses the results obtained.
- Finally, Chapter 6 presents our conclusions about this work and possible directions for future work.

Chapter 2

Context and Related Work

This chapter presents some context that serves as the basis for this work and discusses relevant related work. Section 2.1 presents a study on vulnerabilities in general and a more detailed study on the vulnerabilities addressed in this work. Also, it introduces a description of how these vulnerabilities occur and how they can be corrected and prevented. Section 2.2 analyzes vulnerability detection techniques, namely static and dynamic analysis, fuzzing, and machine learning approaches. Finally, Section 2.3 looks at some tools and techniques used in the subject of automatic software repair.

2.1 Vulnerabilities

Vulnerabilities are the root cause of security problems in software systems. A vulnerability can be described as a flaw or weakness in a system, which can be exploited or triggered by a threat source resulting in a security breach or a violation of the system's security policy [9][10][16]. Vulnerabilities can arise at different stages of a system life cycle but usually result from introduced flaws in the software during the development phases. This issue becomes more evident when programming languages are used that offer more freedom to the programmer and are more error-prone, as is the case with the C programming language. C is a very flexible language that facilitates access to memory in an invalid and unchecked manner. These characteristics lead to a large number of security flaws because programmers assume that the language handles certain aspects when, in fact, it does not. C is very popular and is the language of choice for many applications, although it has characteristics that are commonly misused, resulting in many vulnerabilities in the systems.

Although there are many classes of vulnerabilities to explore and study, the focus of this work is on buffer overflows since they are the root of a large percentage of severe security problems that have emerged over the years [2][3][4][5].

2.1.1 Buffer Overflow

Buffer overflow is probably one of the best-known forms of software security vulnerability. A buffer overflow occurs when a program performs operations outside of the boundaries of the memory allocated to a particular data structure (buffer).

The sample code in Listing 2.1 demonstrates a simple buffer overflow that is often caused by the scenario in which the code relies on external data to control its behavior. The code uses the `gets` function to read an arbitrary amount of data into a buffer. The safety of the code depends on the user to

always enter fewer characters than `BUFSIZE` because there is no way to limit the amount of data read by this function.

```
1 #include <stdio.h>
2 #define BUFSIZE 20
3
4 int main(int argc, char *argv[]) {
5     char buffer[BUFSIZE];
6     gets(buffer);
7     return(0);
8 }
```

Listing 2.1: Buffer overflow example.

Not all buffer overflows lead to software vulnerabilities. However, if an attacker can manipulate user-controlled inputs to exploit a buffer overflow, it can lead to a vulnerability. Even if a buffer overflow cannot be exploited maliciously, it can have unwanted effects on a program. Depending on the size of the overflow and the memory location, a buffer overflow can go unnoticed but can corrupt data, cause erratic behavior, cause the execution of malicious code, or terminate the program abnormally.

The root cause of most buffer overflows is the combination of memory manipulation and wrong assumptions about the size or composition of data. Buffer overflow vulnerabilities usually involve violating the assumptions made by the programmers when using memory manipulation functions that do not perform bounds checking on the buffers on which they operate. Even bounded functions, such as `strncpy`, can cause vulnerabilities when used incorrectly.

Most software developers know what a buffer overflow is, but buffer overflow vulnerabilities are still quite common. Part of the problem is due to the wide variety of ways buffer overflows can occur and the utilization of programming languages like C that are susceptible to them. These programming languages do not perform implicit bounds checking, contain standard functions difficult to use securely, and define strings as null-terminated arrays of characters, which contributes to the susceptibility to the emergence of these vulnerabilities. Some languages have protections that prevent access to memory areas outside of arrays. In the case of Java, when this happens, the exception `ArrayIndexOutOfBoundsException` is raised. With this kind of exception, it is possible to know that there is some operation in the code accessing the memory outside the array, and it becomes easier to find the errors to correct them.

Buffer overflows are not easy to discover, and although they are often easy to correct, they are generally hard to exploit. Nevertheless, attackers have managed to identify buffer overflows in a wide range of software and take advantage of these to perform several types of buffer overflow attacks.

Buffer Overflow Attacks

Buffer overflow attacks can be carried out through different operations and memory areas. It is necessary to know how the memory is arranged to perform this type of attack. The exact organization of process memory depends on the operating system used, but, generically, it is organized into code, data, heap, and stack segments, as shown in Figure 2.1. The code or text segment includes code instructions and read-only data. Typically, this segment is marked read-only, and any attempt to write to it will result in a segmentation fault. The data segment contains initialized data, uninitialized data, static variables, and global variables. The heap is used for dynamically allocating process memory. The stack is a Last

In, First Out (LIFO) data structure used to support process execution by maintaining information that reflects the execution state of the process.

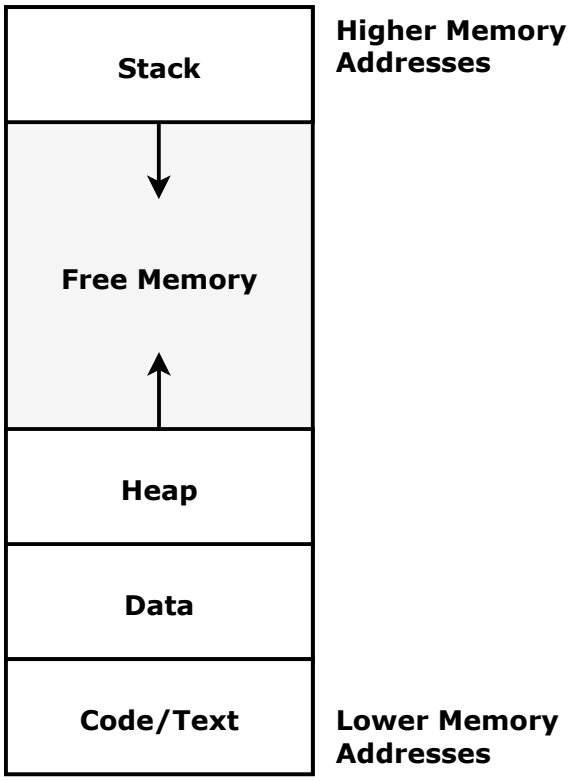

Figure 2.1: Generic process memory organization.

There are a number of different buffer overflow attacks which employ various strategies and target distinct memory areas and pieces of code. However, most of these attacks have their genesis in the two most well-known types:

- **Stack-based buffer overflow** - This is the most common type of buffer overflow attack and involves overflowing a buffer on the call stack. The stack-based approach occurs when an attacker sends malicious data to a program, which it stores in an undersized stack buffer. This operation overwrites the data on the call stack, including the function's return pointer, which is modified by the malicious data so that when the function returns, it transfers control to the attacker's malicious code.

- **Heap-based buffer overflow** - This type of attack targets the heap data and is harder to carry out than a stack-based approach. The heap-based approach occurs when an attacker overflows a buffer that overwrites a pointer to an object allocated in the heap and points it to a virtual table generated by the attacker so that when the program executes it, it runs the code controlled by the attacker.

These are the most common attacks when it comes to buffer overflows, and there are several studies and books that address these attacks and provide detailed explanations of how they can be executed [44][24][17][48]. There are other types of attacks that can be directly or indirectly related to these, such as format string, integer overflow, and off-by-one error attacks [57].

Buffer Overflow Protections

There are different solutions for protecting against buffer overflows. Programmers can prevent buffer overflows by building security measures into their code, such as avoiding standard library functions that do not perform bounds-checking and never trusting user input. Regularly testing code and using programming languages that include built-in protection can be other measures that help mitigate these vulnerabilities. However, the latter may not be feasible as some languages are more suitable for certain purposes than others.

Currently, modern operating systems have runtime protections and exist some mechanisms that enable additional security against buffer overflows, such as the following ones:

- **Canaries** - Allow the detection of stack corruption. The canaries are special values placed before (or after) the memory locations intended to protect. When accessing the protected memory, the canaries are checked to confirm that the value did not change and the memory has not been corrupted.
- **Address Space Layout Randomization (ASLR)** - ASLR randomly rearranges the starting address of the address space segments. Typically, buffer overflow attacks need to know where the executable code is located, and randomizing address spaces makes this nearly impossible.
- **Data Execution Prevention (DEP)** - DEP marks certain memory areas as either executable or non-executable, preventing an exploit from running code found in a non-executable area.
- **Structured Exception Handler Overwrite Protection (SEHOP)** - SEHOP helps stop malicious code from attacking the Structured Exception Handler (SEH), a built-in system for managing hardware and software exceptions. Attackers may look to overwrite the SEH by performing a stack-based overflow attack to overwrite the exception registration record, which is stored on the program's stack.

These are just a few protections that can be used to protect against buffer overflow attacks. There are many others, some that are inspired by these methodologies and some that implement innovative methods. Younan et al. [57] performed a survey on C and C++ code injection. In this study, the authors analyzed buffer overflow vulnerabilities and the attacks that can be executed to exploit them. Also, they explain several existing countermeasures that can be used against different types of buffer overflow attacks. Piromsopa et al. [45] also conducted a survey in which they addressed various protection approaches against buffer overflows and categorized the approaches into different categories.

Even though all these protections exist, they are not enough. New buffer overflow vulnerabilities continue to be discovered and exploited. When new vulnerabilities are discovered, the engineers or the companies must react quickly to patch the affected software and ensure that the users access the patch and use it.

2.2 Vulnerability Detection

The vulnerability detection process is difficult to perform and requires a large amount of time. The search for processes that can help programmers accomplish this task easier and faster is a major focus of research

today. The number of studies to create automatic methods to achieve this goal has increased significantly in recent years, and a variety of techniques are being used, such as static and dynamic analysis and fuzzing. In addition, some research seeks to use machine learning techniques to find vulnerabilities in the code, taking advantage of models to predict where vulnerabilities may exist and thus make the detection process faster.

2.2.1 Static Analysis

Static analysis has the objective of analyzing the source code of an application to find bugs. Static analysis methods allow the analysis of code without any execution of it. Thus, it is possible to use these methods at any stage of an application development process, even if it is not complete.

Static analysis tools are widely used to help in software development because manual code auditing becomes very time-consuming and usually infeasible when the size of the source code increases, so an automatic process helps a lot. This type of tools allow to analyze the code faster than manual auditing and obtain some level of abstraction. Some only detect vulnerabilities in some functions, others anywhere in the code. Some do their analysis one function at a time; others analyze a complete program.

Although it has many qualities, this kind of analysis cannot solve all problems. These tools analyze the code looking for patterns that may indicate vulnerabilities. There must be an update of the rules and patterns they should look for to find new vulnerabilities. If the rules are not updated, there may be many vulnerabilities that are not found, i.e., there are a high number of false negatives. There may also be false positives because of problems that can be detected that are not vulnerable. For this reason, one of the problems with these methods is their imprecision. Therefore, it is necessary to audit the results manually to solve these problems and improve the accuracy of the results.

Within static analysis, there are different types of mechanisms. The most known are the following ones:

- **Lexical analysis** - The lexical analysis approach compares resulting tokens from a preprocessing and tokenization of source files with some known vulnerable constructs. This approach produces plenty of false positives due to the lack of knowledge of the target's semantics.
- **Control flow analysis** - The control flow analysis methods try to follow the control path of the program through rule checks and control graphs that are generated for the functions.
- **Data flow analysis** - Data flow analysis methods try to understand how data moves within programs, namely from the moment they enter the program to the places where they are used, possibly in dangerous instructions. In this category, the most common is the method of taint analysis, which uses type qualifiers – tainted and untainted – to perform taint analysis. This method uses type inference rules to detect vulnerabilities through function annotations that determine whether they return tainted data or require untainted data.

Boudjema et al. [19] presented a tool based on static analysis methods, more specifically, abstract interpretation extended with security vulnerability property checks to detect security vulnerabilities in C applications automatically. They verify the properties by analyzing the language specification and documentation of the main language libraries. To locate the vulnerabilities, they define properties related

to different classes of vulnerabilities, namely format string, command execution, and buffer and memory vulnerabilities. They present a detailed description of the properties and possible attack scenarios that can be performed to exploit the vulnerabilities addressed. Although some of the proposed properties do not have many test cases, they have shown that it is possible to detect vulnerabilities in an automated way through the described properties with an acceptable number of false positives and false negatives.

J. J. Kronjee [36] explored machine learning and static code analysis techniques to detect vulnerabilities in PHP applications. The work presents a detailed study of several vulnerabilities and machine learning methods. He used control flow graphs and abstract syntax trees to extract features for use in machine learning modules. Also, he used taint analysis to determine possible vulnerable paths. He constructed a data set by combining mined samples from the National Vulnerability Database and used it to train and test the classifiers. He demonstrated that machine learning techniques in combination with features extracted from control flow graphs and abstract syntax trees can be used for vulnerability detection in PHP applications.

Flynn et al.[27] used alerts from multiple static analysis tools to develop a classifier to reduce analyst effort and remove flaws. They used multiple static analysis tools to generate outputs about the inspected code. Then they used an enhanced version of an existing tool to aggregate and evaluate the previous results. This version of the toll produced alerts that were mapped to a CERT Secure Coding Rule [12] to facilitate auditing of the alerts. After the alert's consolidation, a group of auditors classified it to determine the false or true positives. This analysis, alongside other coding rules, was processed into a training data set that was used to construct prediction models. The results showed that it is difficult to consolidate different data to use in the classifiers, which limits their performance.

Yamaguchi et al. [56] proposed a method for assisted discovery of vulnerabilities in source code. Their goal was to create a method to make manual auditing more effective, helping and guiding the inspection of the source code. They created a method that places the code in a vector space so that typical Application Programming Interface (API) usage patterns can be determined automatically. To capture API usage patterns and transfer these known vulnerability patterns to other pieces of code, they combined static code analysis and machine learning techniques. These patterns implicitly capture the semantics of the code and allow extrapolating known vulnerabilities, identifying potentially vulnerable code with similar characteristics. This extrapolation process serves as a guide for the analyst and facilitates the inspection of the source code. Many vulnerabilities can be captured by API usage. However, there are also cases where the code structure is more relevant for auditing.

2.2.2 Dynamic Analysis

Dynamic analysis, opposingly to static analysis, analyzes programs while they are running. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to cover all the possible outputs. Dynamic code analysis tools allow to perform an analysis and identify potential issues that arise during the actual execution of the program and impact its reliability.

Haugh et al. [33] presented a tool, STOBO, for testing C programs for buffer overflows using dynamic analysis. This tool analyzes some functions considered functions of interest that can cause buffer overflows. The idea is to verify if the buffers are static or dynamic allocated. Then verify if the call to the functions that use those buffers is safe, checking if they follow some rules regarding the size of the

buffers. The tool generates different warnings depending on the buffers allocation method. This tool only detects vulnerabilities that may derive from the misuse of library functions. The results showed that the tool can find vulnerabilities but generates some false positives.

Yun et al. [58] presented an automatic tool to discover heap exploitation primitives, ARCHEAP. The main idea of ARCHEAP is to explore spaces automatically by specifying a set of modern designs and the root causes of vulnerabilities as models, using heap operations and attack resources as actions. Their goal is that, during execution, ARCHEAP checks whether the given combinations can be used to build primitive exploits, such as arbitrary write or overlapped chunks. They analyze several heap allocators, current exploits, and the types of associated bugs. Their solution uses an American Fuzzy Lop (AFL) extension to perform the random heap actions. The results showed that they were able to find new exploration techniques, which shows that the methodologies used perform well in discovering new exploits.

Another approach that can be used to try to increase the number of vulnerabilities found and speed up the process is to combine several tools. Vorobyov et al. [54] conducted a study to compare three different runtime verification tools for the C language. They aimed to give a realistic comparison between the tools and focus on testing runtime tools. Another criterion of choice was tools with different approaches, so they chose a formal semantic-based tool, a formal specification verifier, and a memory debugger to evaluate the detection power of these. They intended to find the cumulative detection ratio of the tools used together, the detection ratio of each one of the tools, and find if they are complementary to each other in finding vulnerabilities. The results of the tests showed that the tools used together can find even more vulnerabilities because they complement each other.

2.2.3 Fuzzing

Fuzzing is a popular software testing method that injects random inputs into a system to reveal software defects and vulnerabilities. A fuzzing tool injects these inputs into the system and then monitors for exceptions such as crashes or information leakage.

Most fuzzers differ in many significant ways, but, in general, they all follow a simple procedure. They receive an initial input and run the program with it. After running the program with that input, they mutate it to generate new input, which might lead to different paths coverage when it is executed. Some fuzzers also use the information gathered in the execution to help generate better program inputs. If the program input is deemed interesting, it is saved to be further mutated to uncover different paths in the program. This process is repeated until something ends the execution of the fuzzer, which is generally accomplished by a timeout or by reaching a certain number of discovered bugs, with the ultimate goal of trying to find inputs to make the program crash. In the end, these inputs are returned to the user, which can use them to reproduce the crash.

Fuzzers can be categorized according to the following criteria:

- Method of input generation;
- Awareness of input structure;
- Awareness of program structure.

Depending on whether inputs are generated, a fuzzer can be generation-based or mutation-based. The generate-based fuzzer generates the input data from scratch without relying on previous inputs. The mutation-based fuzzer generates the input data based on defined patterns by mutating the provided seeds.

In terms of the awareness of input structure, if the fuzzer has no awareness of the format of the input data, it is considered a dumb fuzzer. Dumb fuzzers produce random input that does not necessarily match the correct format. If the fuzzer is aware of the format of the input data, it is considered a smart fuzzer. A smart fuzzer leverages the input model to generate a greater proportion of valid inputs.

Regarding the awareness of program structure, a fuzzer can be one of the following types:

- **Black-Box** - The fuzzer treats the program as a black box, i.e., without having any knowledge about the source code of the program. It can execute several hundred inputs per second, can be easily parallelized, and can scale to programs of arbitrary size. However, it can take an extremely long time to find deeply nested bugs due to the random nature of the input generation, which provides limited coverage and so the testing can be inefficient.

- **White-Box** - The fuzzer has access to the program's structure and symbolically executes the program under test, gathering constraints on inputs from conditional branches encountered throughout the execution. It has a greater coverage when compared with other types. However, the time used for analysis can become excessive.

- **Gray-Box** - The fuzzer uses lightweight instrumentation to obtain information program structure without requiring any previous analysis. This may cause a significant performance overhead but increases the code coverage as a result.

Since fuzzing was introduced, it is used for security testing and quality assurance proposes [43][52]. Several studies have been conducted, and new techniques have been discovered while others are being improved. There is a great diversity of work related to fuzzers and the different types of fuzzers. Woo et al. [55] developed an analytical framework using a mathematical model of black-box mutational fuzzing. Bounimova et al. [20] showed the results of the white-box fuzzer, SAGE [30]. They describe the challenges with running the fuzzer in production and show data on the performance of constraint solving and dynamic test generation. Chen et al. [21] described a directed gray-box fuzzer, Hawkeye, that combines static analysis and dynamic fuzzing. More recently a new approach to using fuzzers has emerged. Chen et al. [22] studied the performance of an ensemble fuzzing approach. The results obtained were very promising since trough this work it was possible to discover several new vulnerabilities. Haller et al. [32] created Dowser, a guided fuzzer that combines taint tracking, program analysis, and symbolic execution to find buffer overflow and underflow vulnerabilities.

LibFuzzer [11] is a coverage-guided, evolutionary fuzzing engine to test C/C++ software. LibFuzzer works by implementing a fuzz anchor. An anchor is a program written in C or C++ that allows the tester to specify a fuzzing entry point. This entry point is a function that accepts data and the size of the data. With this function, the tester can direct the fuzzer to whatever desired function, where it will then execute the fuzz target. This type of fuzzers require some sample inputs for the program under test. LibFuzzer generates random mutations based on the input given originally. If the fuzzer discovers new and interesting test cases, they are saved for later usage or mutation. Honggfuzz [8] is a similar fuzzer. It is a security-oriented, feedback-driven, evolutionary fuzzer.

2.2.4 Machine Learning

Machine learning approaches are being used to automate many different tasks. Recent advances in this area have resulted in a wave of interest in their application for automatic vulnerability detection. Existing work that attempts to achieve this goal is usually based on a process that involves performing the following tasks: extracting training data from available source code, using this data to train a classifier, and finally, using this trained classifier to predict where vulnerabilities may exist in new test cases. Below is some related work that uses machine learning techniques to detect vulnerabilities.

Russell et al. [46] presented machine learning techniques for automated detection of vulnerabilities in C/C++ source code. They built a source code dataset with functions from GitHub and Debian repositories. Additionally, they made a lexer to create a representation of source code ideal for machine learning training. To label the vulnerabilities, they used static analyzers because of the better performance compared with dynamic analysis and commit-message-based labeling. Their approach combines neural feature representation with a random forest classifier. The methods presented do not require code to be compiled to look for vulnerabilities and perform better than other static analysis tools. A disadvantage of this machine learning method could be that it does not provide clear information about the localization of the vulnerabilities in the code.

Grieco et al. [31] conducted a study to predict if a test case is likely to discover software vulnerabilities by using lightweight static and dynamic features implemented using machine learning techniques. They do this mostly by analyzing binary programs according to some procedure to perform the vulnerability discovery. The goal was to train a classifier through these features and supervised learning techniques. The procedure for detecting vulnerabilities comprises two components: a fuzzer that mutates the test cases and a module that dynamically detects exploitable memory errors. Their proposed methodology works in two phases. A training phase in which they train the tool, and the collection phase in which a trained classifier is used to predict whether or not new test cases will find bugs, which can then be prioritized for further analysis. The evaluation results show that by analyzing a small percentage of the test set pointed as potentially interesting, the tool can predict with reasonable accuracy which programs contain a vulnerability, which results in a significant increase in the fuzzing speed.

Li et al. designed and implemented a deep learning-based vulnerability detection system called VulDeePecker [37]. The authors discuss some guiding principles for using deep learning techniques to achieve the intended objectives, such as the representation of the programs, the determination of granularity in which the detection process should be conducted, and the selection of specific neural networks for vulnerability detection. They propose the use of code gadgets, a few lines of code that are semantically related to each other and can be vectorized as input for deep learning to represent programs. Their approach goes through two distinct phases. The first one is the learning phase, where patterns of vulnerabilities are generated. The second phase consists of the detection phase that uses the patterns acquired in the previous phase to determine whether a program is vulnerable or not and indicate the localization of the vulnerability, if any. The results obtained showed that the VulDeePecker could achieve a lower false-negative rate than other vulnerability detection systems and was able to find vulnerabilities that other systems could not.

Dahl et al. [25] explored machine learning techniques for vulnerability detection. They used neural networks to identify the presence of potential stack-based buffer overflow vulnerabilities in assembly

code. They assumed that code could be treated as a form of language and processed it using recurrent neural networks based on long short-term memory cells.

Zhou et al. [59] proposed Devign, a general graph neural network-based model for graph-level classification through learning on a rich set of code semantic representations. It was inspired by the fact that vulnerability patterns manually crafted with the code property graphs, integrating all syntax and dependency semantics, have been proved to be one of the most effective approaches to detecting software vulnerabilities. Devign automates this process on code property graphs to learn vulnerable patterns using graph neural networks.

2.3 Automatic Software Repair

Automatic software repair is one of the most widely discussed topics nowadays. It consists of automatically finding a solution to fix software bugs without human intervention. This subject is of great importance, as it can help solve many problems, which are becoming more and more common due to the growth of software usage daily. However, it is a very challenging topic because fixing bugs is not an easy task. Several techniques have been extensively investigated as solutions for efficiently repairing and maintaining software in the last few years. Recent work gathered and organized the body of knowledge about automatic software repair [29][41].

2.3.1 Repair Approaches

Automatic software repair approaches detect software failures and perform corrective adjustments over the targeted program to fix them, i.e., software repair or restore its normal execution. They are divided into two main approaches:

- **Software Healing (or State Repair)** - Software healing consists in changing the state (e.g., stack, heap) of the program under repair.

- **Software Repair (or Behavioral Repair)** - Software repair consists in changing the program's behavior by altering the code. It can be done offline or at runtime.

These approaches perform two distinct processes:

- **Healing process** - It is composed of two steps that might be executed iteratively. The first one is the healing step which consists of executing a healing operation that can prevent or mitigate a failure that has been detected. The second one is the verification step which consists of checking if the application is running as expected after the healing operation has finished.

- **Repairing process** - It is composed of three steps that might be executed iteratively. The first one is the localization step, which identifies the locations where a fix could be applied. The second one is the fix step, which generates fixes that modify the software in the code locations returned by the localization step. The third one is the verification step, which checks if the synthesized fix has actually repaired the software.

2.3.2 Repair Techniques

The repair techniques are categorized into generate-and-validate and semantics-driven approaches. The generate-and-validate approaches define a search space that is explored for potential solutions. The semantics-driven approaches encode the problem of repairing a program as a formula whose solutions correspond to the possible fixes of the program under repair or as an analytical procedure whose outcome is a fix.

Software repair techniques have been evaluated in many diverse contexts, including papers presenting new approaches and independent empirical evaluations. The following are some tools that are somehow related to automatic software repair.

PASAN [51] is a technique designed to repair buffer overflow vulnerabilities. It works by first detecting the inputs used in control-hijacking attacks and then using these inputs to generate fixes that remove the vulnerabilities exploited in the attacks. PASAN instruments the application under repair to extract information about the size of static arrays and dynamically allocated buffers. It uses different strategies to produce a candidate fix. PASAN attempts to find the library function or the loop that originated the problem and change it by introducing appropriate checks in the code. The generated fix is tested by replaying the attack against the fixed program.

AutoPAG [38] aims at reducing the time needed for software patch generation. It first instruments the application to detect the variables that overflow and thus identify the tainted sets of statements and variables. The fix generator works on these tainted sets using different fix templates: redirecting an out-of-bound read within the buffer boundary, replacing a call to a function that allows out-of-bound writes with a call to a safe function, and simply skipping the statement that causes the out-of-bound violation. The fixed application is then tested against the same out-of-bound exploit that triggered the repair process.

Code Phage [50] targets buffer overflow problems. It uses a set of donor programs to extract the conditions that should be added to the program under repair to prevent the buffer overflow. The set of donor programs must be programs that implement the same functionality of the program under repair. This repair technique assumes that there might exist a donor program that contains the check that is missing in the faulty program. Then it can repair the fault by copying the check from the donor to the program under repair.

Gao et al. [28] presented, BovInspector, a tool that uses static analysis and symbolic execution to analyze buffer overflow vulnerabilities and suggests fixes by applying the following three change strategies: add boundary checks, replace danger function calls with calls to safer functions, and modify buffer instantiation. The specific change to be applied to each function call is based on a set of patterns.

Ding et al. [26] characterized buffer overflow vulnerabilities in the form of four patterns and proposed ABOR, a framework that integrates and extends existing techniques to remove buffer overflow vulnerabilities. ABOR consists of two modules: vulnerability detection and vulnerability removal. ABOR works iteratively: once the vulnerability detection module captures a vulnerable code segment, the segment is fed to the removal module; the fixed segment will be patched back to the original program. ABOR repeats the above procedure until the program is buffer overflow-free.

Shahriar et al. [49] proposed a set of general rules to address the mitigation of buffer overflow vulnerabilities for C/C++ programs. The authors developed a set of rules to identify vulnerable code and

how to make the code vulnerability free. They proposed 12 patching rules to replace vulnerable code and remove buffer overflows at the application unit level. The proposed rule-based approach addresses both simple and complex forms of code that can be vulnerable to buffer overflows ranging from unsafe library function calls to pointer usage in control flow structures. The results showed that the proposed rules could identify previously known buffer overflow vulnerabilities and also find new ones.

Angelix [40] is a semantics-driven repair technique that aims at synthesizing multi-line fixes while preserving scalability. To generate multiline fixes without sacrificing scalability, Angelix exploits the concepts of angelic path and angelic forest. An angelic path encodes part of the repair problem as a set of triples each one containing an instance of a suspicious expression, its angelic value, and its angelic state. Angelix paths are extracted using symbolic execution. An angelic forest fully encodes the repair problem as a set of angelic paths. The angelix forest is fed to a fix synthesis engine to produce multi-line fixes.

Morgado et al. [42] proposed an approach for automatic code correction of PHP web applications. They focused on the study of injection vulnerabilities and the existing forms of sanitization for these cases and their defects. The developed tool can determine the most suitable correction for each type of bug and where to apply it. Their approach aims to fix applications by inserting new lines of code at the entry points to validate inputs that are used later in sensitive sinks. To know which correction to apply and where to apply it, they combine taint analysis techniques with variable simulation to keep track of the variables. To test the developed tool, they used real applications vulnerable to the functions covered in the study. The results showed that it was able to apply a safe correction in most cases, preserving the validity and correct behavior of the programs.

In recent years, advancements in machine learning brought a wave of progress within the field of software repair. This trend leads to the appearance of different repair techniques named data-driven approaches. A data-driven approach is when decisions are based on analysis and interpretation of hard data rather than on observation. A data-driven approach ensures that solutions and plans are supported by sets of factual information. Below we present some of the related work that uses machine learning approaches to repair software.

Chen et al. [23] explored and developed an approach to perform code repair that automatically generates patches for security vulnerabilities. To achieve this objective, they used a sequence-to-sequence (seq2seq) machine learning technique with byte pair encoding that learns the mapping between two token sequences of source code. They used data collected from GitHub commits that presented the vulnerable and corrected code to create the training dataset, in which they chose C functions that the seq2seq algorithm could use and divided them by different sizes. Their results showed that the seq2seq algorithm performance is low, fixing general vulnerabilities and depends a lot on the size of the inputs used in the tests. However, they proved that it is possible to fix vulnerabilities in an automated way.

Bader et al. [18] presented Getafix, an approach that aims to produce human-like corrections and, at the same time, be able to propose corrections in time proportional to what would take to obtain static analysis results. The approach first divides a certain set of example corrections into Abstract Syntax Trees (ASTs). Then it extracts correction patterns from these ASTs, based on a new hierarchical clustering technique that produces a hierarchy of correction patterns ranging from very general to very specific corrections. Finally, given a bug to correct, Getafix finds appropriate correction patterns, classifies all

candidate corrections, and suggests the main solutions for the developer. As a check during the third step, Getafix validates each suggestion through a static analyzer to ensure the correction removes the warning. Getafix uses a simple but effective classification technique that uses the context of a code change to select the most appropriate fix for a given bug. The idea is that the tool learns through previously created corrections to create new ones. The results showed that the tool can perform well and accurately predicts fixes for several bugs, reducing the time developers spend fixing recurring bugs.

Vasic et al. [53] presented an approach that jointly learns to localize and repair bugs. The model classifies the program as faulty or correct, locates the bug when the program is faulty and applies a fix to it. To solve the problem of classification, location, and repair, they used a multiheaded pointer network architecture where one pointer head points to the faulty location and another to the location where the correction should be made. They compared a pointer network on top of a neural network to a graph neural network and observed that their solution achieved better results. Also, compared the jointly model with an enumerative approach, and the results showed that the model outperformed the enumerative approach by using a model that can predict a fix given the location of a bug. They concluded that the solution, despite its limitations, can perform well compared to some approaches for similar purposes.

Sawadogo et al.[47] proposed an approach to catch security patches as part of an automatic monitoring service of code repositories. This work was motivated by the delay between the release of a security patch and its application. To differentiate between security patches and others, they used commit log and code analysis to collect data for the binary classification task. After that, they carried out a feature engineering step in which they reduced the volume of data collected only to the essential and transformed this data into numerical vectors to use in the learning algorithms. They opted for the use of a co-training algorithm because of the lack of labeled data, which proved to be the best option once the proposed approach demonstrated high precision and recall presenting a significant improvement over the state-of-the-art.

Lutellier et al. [39] proposed CoCoNuT, a new generate-and-validate technique that uses ensemble learning on the combination of convolutional neural networks and a new context-aware neural machine translation architecture to automatically fix bugs in multiple programming languages. They introduced a new context-aware neural machine translation architecture that represents the buggy source code and its surrounding context separately. CoCoNuT also takes advantage of the randomness in hyperparameter tuning to build multiple models that fix different bugs and combines these models using ensemble learning to fix more bugs.

Jiang et al. [35] presented CURE, a neural machine translation-based technique with three major novelties. First, CURE pre-trains a programming language model on a large software codebase to learn developer-like source code. Second, CURE designs a new code-aware search strategy that finds more correct fixes by focusing on compilable patches and patches that are close in length to the buggy code. Finally, CURE uses a subword tokenization technique to generate a smaller search space that contains more correct fixes.

Chapter 3

Proposed Solution

This chapter describes the main challenges for identifying and fixing buffer overflow vulnerabilities in C programs and verifying the effectiveness and correctness of the fixed code in an automated manner. Also, it presents our proposed approach to address these challenges. Section 3.1 explains the challenges this work faces and the reasoning behind how we want to manage them. Section 3.2 presents an overview of the proposed solution architecture and a brief description of its modules. Section 3.3 describes in detail the key modules of the architecture. Finally, Section 3.4 specifies and analyzes the sensitive sinks related to buffer overflows addressed by our approach.

3.1 Challenges

This section introduces some problems from which arose a set of challenges that must be solved to create an automated vulnerability discovery and repair process, validating the results obtained for each phase. Additionally, we present the key ideas that emerged to address these challenges and their reasoning.

3.1.1 How to find vulnerabilities and ensure that they are exploitable?

One of the problems related to the identification of vulnerabilities in code through static analysis tools is the existence of false positives. Many tools generate several false positives, making it difficult to locate real vulnerabilities in the source code. False positives can derive from a wrong inspection of the code made by static analysis tools and from a correct identification of a vulnerable execution path, which starts at some entry point and ends at a sensitive sink, but there is no input able to exploit such alleged vulnerability. Therefore, it is necessary to ensure that the vulnerabilities found statically are actually exploitable, giving evidence of such by providing one exploit at least.

Our idea to overcome this problem is to analyze the code of a program through static analysis to discover vulnerabilities, whose results will be candidate vulnerabilities, and so can have false positives. Next, we will use fuzzing techniques to filter these results and check whether these techniques can exploit the identified candidate vulnerabilities, producing their exploits.

3.1.2 How to generate executable and compilable code slices from vulnerabilities found statically?

Fuzzing techniques to exercise the program under test demands that the program is running, so an executable file of it is required. On the other hand, static analysis tools do not provide executable or

compilable code as output for vulnerabilities they report. Most of them only output the line of the code where an entry point reached the sensitive sink. Very few tools return the complete slice of the vulnerable code, i.e., the lines of the vulnerable execution path that starts at an entry point and ends at a sink, but this code is neither compilable nor executable. Therefore, the challenge here is how to capture all the code needed for each candidate vulnerability reported by static analysis and make it compilable and executable to be exercised by fuzzing.

The main idea is to generate a small program for each potential vulnerability found, composed of the slice of the vulnerable code and all the other code necessary to make it compilable and executable, such as the main function, libraries, and variables declaration. For that, we will employ code parsing techniques over the output of static analysis and the files of the program under test to capture all code required.

3.1.3 Where and how to correct the code?

One of the main challenges of automatic code correction is to decide where the code for removing the vulnerability – fix – should be inserted. This decision is difficult because a slight change in the code may alter the program's logic, which its effect needs to be avoided to maintain the correct program's behavior. In our case, since we want to fix vulnerabilities associated with buffer overflows, which are usually related to sensitive sinks, our focus will be on fixing the lines of these sinks. Hence, we propose the insertion of fixes in these lines or close to them.

Another challenge associated with code correction is to decide what type of correction to apply in each case, since a given vulnerability class can be expressed in the code in different forms, even for the same sensitive sink. Moreover, as there is no universal fix for all cases and each sensitive sink is used differently with distinct arguments, it makes the process of correction more difficult. Our idea is to fix the issues associated with specific sensitive sinks. For some sensitive sinks, the fix may be to replace them with their secure version (e.g., `strncpy` for `strcpy`), but some may have no possibility of doing this because they do not have a safe version (e.g., `scanf`). But in both cases, our approach will capture the sinks' arguments and whether they need some validation before using them in the sinks, and determine the correct amount of bytes that must be used by the fixes.

In sum, our conception to solve these two challenges is to construct a set of fix templates that will be used dynamically when the vulnerable code is being inspected to determine what is necessary to fix and where the fix will be inserted. Based on these inspections, the fix template is selected and generated the final fix. Some fixes will replace sensitive sinks with their safer version if they exist. Otherwise, fixes will modify the sink statement or insert code instructions close to it to ensure that the sensitive sink is used correctly and safely.

3.1.4 How to know that the fix applied is effective?

An issue associated with automatic code correction approaches existing in the literature is that there is no automated process to verify the effectiveness of the fix once it is applied, i.e., if it does not spoil the correct behavior of the program and its logic, and if it actually removed the vulnerability. This process has to be done manually by programmers.

We propose to automate this validation process by using the exploits generated in the fuzzing task during the vulnerability verification process. The validation process will use these exploits that break the functioning of the original program to verify that the fix works, i.e., if it cannot crash the program's operation, it means that the applied fix effectively removed the vulnerability and, hence, corrected the code. Also, if it cannot hang the program's behavior and logic, it means that the fix was correctly generated syntactically and inserted in the right places in the code. In addition, our validation process will generate new test cases to try to circumvent the fix and spoil the program's behavior.

3.2 Approach Overview

This section describes the architecture of our approach to identify vulnerabilities, fix them, and verify the fixes' effectiveness. The key idea is to combine static analysis with fuzzing to identify real vulnerabilities with higher precision and evidence of their exploitation, and validate the correction made. Also, since fuzzing has the goal of covering all code of the application, which task can take a considerable amount of time, we opt for fuzzing small programs for each potential vulnerability found by static analysis. These small programs will only contain an execution path that will be quickly exercised by fuzzing. Figure 3.1 illustrates the approach's architecture with its main modules. The architecture consists of the following elements:

- **C/C++ Program** - The C/C++ program files that we want to test and correct, which can contain one or more vulnerabilities.

- **Vulnerability Identifier** - This module is responsible for identifying possible candidate vulnerabilities in the received program. It uses static analysis techniques to collect information about potential vulnerabilities and their location in the program, namely the respective line number in the file. Using this information, it generates slices of the vulnerable code from the entry point to the sensitive sink.

- **Executable Generator** - This module receives the vulnerable slices of code from the previous module. To generate an executable file for each candidate vulnerability found, it uses the slice received and adds from the program files other instructions needed to obtain a compilable file. Afterward, this file is compiled, instrumentalized, and generated its executable that is forwarded to the Program Validator.

- **Program Validator** - This module uses fuzzing techniques for validating the code received from the Executable Generator in two distinct phases. Validation is performed in the first phase to exploit the candidate vulnerabilities found by the vulnerability identifier and generate thus the exploits for them. For those vulnerabilities it cannot exploit, they are marked as possible false positives. The remaining ones, i.e., the exploitable vulnerabilities, are signalized as such and their exploits stored for the second phase. The second phase uses the previously generated exploits to verify if the fixes applied are effectively safe. Also, it mutates the exploits to check if there are new exploits that can break the fixes and that the application does not hang.

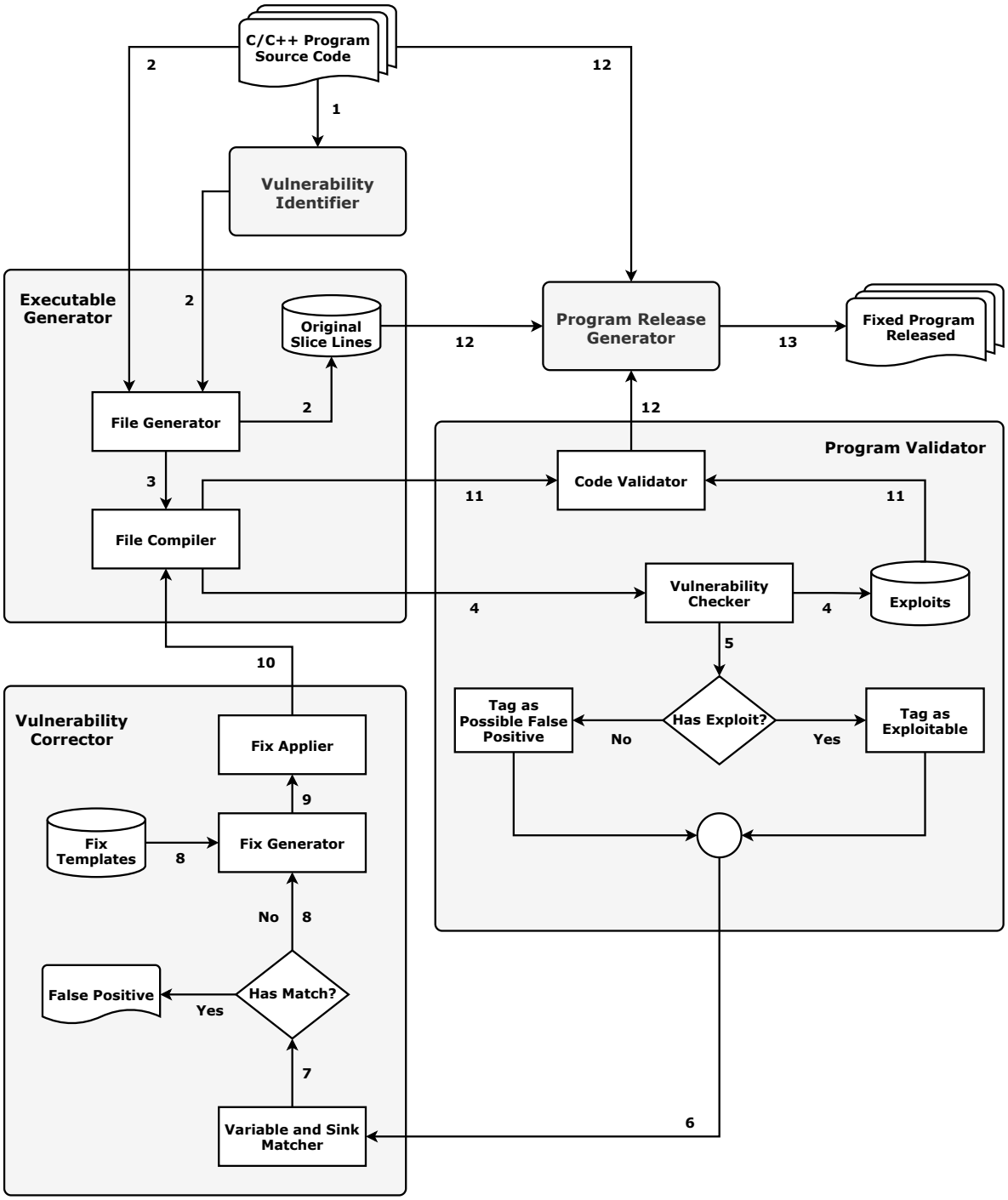

Figure 3.1: Overview of the approach's architecture.

- **Vulnerability Corrector** - This module analyzes the received code from the Program Validator (first phase), identifies the existing sensitive sinks, and determines the variable sizes of the arguments of the sinks. After this analysis, it checks for the possibility of buffer overflows through the size of the variables used in the sensitive sinks. If it verifies that such a possibility exists, it uses the fix template indicated for that sensitive sink to create a fix for that vulnerability and applies it to the code. Also, it attests if the code signalized as possible false positive or as exploitable is it, reporting as false positive the former and proceeding with the code correction for the latter. In

addition, the corrected code follows to the Executable Generator to produce its executable and then to the Program Validator to proceed with the second phase of validation.

- **Program Release Generator** - This module is responsible for applying the resulting and validated fix to the respective lines of the original program files. The output of this module is a new release of the program with its files containing the corrected code, i.e., with the vulnerabilities fixed.
- **Fixed Program Released** - New version of the program files with the vulnerabilities fixed.

3.3 Main Modules

This section describes in more detail how the main modules of the approach's architecture work and how they interact with each other.

3.3.1 Vulnerability Identifier

The first fundamental process to build a tool for our purpose is to locate the potential vulnerabilities, and the Vulnerability Identifier is the module responsible for performing that action. It receives a C/C++ program to be analyzed and scans it for potential vulnerabilities related to sensitive sinks with known problems associated with buffer overflow risk. It works in two steps to identify possible candidate vulnerabilities and extract their vulnerable code slices. A slice is an execution path that starts at an entry point and ends at a sensitive sink. Between the entry point and the sensitive sink, the slice contains all instructions and variables dependent on them.

First, it scans the source code of the program under test, looking for sensitive sinks that possible entry points can reach. This scan results in a list of hits that contain the potential vulnerabilities and their location in the file, i.e., the line of the code where the sinks' instructions are.

In the second step, for each hit, it generates the slice containing all instructions associated with it. The slice starts with the hit instruction, i.e., the sink instruction. Next, it analyzes this instruction and collects information about the variables used in the sink. After knowing which variables are used, the received program is parsed to gather the lines associated with these variables. To do so, after the parsing of the source code, the Vulnerability Identifier performs a bottom-up approach for tracking the variables and the ones dependent on them to where they are declared and initialized, and then extracts all lines associated with them. The resulting lines of code are combined with the sensitive sinks' line, generating a slice, which is stored in a file and forwarded to the next module.

Listing 3.1 shows a buffer overflow example of a hypothetical program to be analyzed.

When scanning this code, the Vulnerability Identifier outputs line 6 as being a potential buffer overflow. Next, it obtains the variable argument from the `scanf` sink, i.e., the buffer variable, and then goes up along the code until finding line 4, where the buffer variable is declared. At the end of this process, these two lines are combined, generating the slice of vulnerable code that is stored in a file. Listing 3.2 illustrates the slice of code generated from the code of Listing 3.1.

At the end of this module, we will have a set of slice files according to the number of vulnerabilities found by the static analysis phase. The objective of having this set is to expedite the way of exploiting the vulnerability by fuzzing (see Program Validator - Section 3.3.3).

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     char buffer[15];
5     printf("Enter a string\n");
6     scanf("%s", buffer);
7     printf("Entered string: %s\n", buffer);
8     return(0);
9 }
```

Listing 3.1: Example of a C program containing a buffer overflow.

```
1 char buffer[15];
2 scanf("%s", buffer);
```

Listing 3.2: Slice of code vulnerable to buffer overflow extracted from Listing 3.1.

3.3.2 Executable Generator

The goal of this module is to create a complete program syntactically correct and compilable, for each slice file that contains a potential vulnerability that we intend to correct. Hence, besides holding the vulnerabilities, each file has to be completed in order to be compilable and executable to be used in the subsequent phases of the approach pipeline. Therefore, it is possible to divide this module into two sub-modules: File Generator and File Compiler.

File Generator

This sub-module receives the slices of code generated by the Vulnerability Identifier, but this code is neither compilable nor executable. It parses the received slices of code and collects information about the sensitive sinks and variables used. Next, it statically analyzes the code of the original program to extract other necessary data, namely constants, directives, and other needed functions, intending to get a functional program file. Once this information is obtained, it is added to the previously received slices, which then contain the lines required to make the slices conform to the language's syntax.

After this task, the resulting file contains a `main` function to be executed, the necessary libraries, and the slices with all the required information for the file to be compiled. In addition, this sub-module registers the line numbers of the original file corresponding to the slice lines to be used later by the Program Release Generator (see Section 3.3.5). Finally, this new file is passed to the File Compiler sub-module to proceed to the next stage of the process. Listing 3.3 illustrates the file generated from the slice of code shown in Listing 3.2.

As we can observe, the slice of Listing 3.2 was completed by adding the `main` function, the `stdio.h` library, and the `return` instruction turning thus the file compilable.

File Compiler

This sub-module produces the executable file of each potential vulnerability file, completed by the previous sub-module, and, therefore, prepares this file for the validation process. It works in two distinct

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     char buffer[15];
5     scanf("%s", buffer);
6     return(0);
7 }
```

Listing 3.3: File generated from the slice example from Listing 3.2.

phases in our approach, since the Program Validator module performs two distinct tasks with two different versions of slices under test. Section 3.3.3 describes these validation phases in more detail.

Although the File Compiler works in distinct phases, the tasks it performs are always the same. It compiles and instrumentalizes the previously generated file according to the requirements for using the fuzzing techniques in the validation process. However, in the first phase, the file it compiles contains the potential vulnerability we want to test and fix, while in the second phase, the file it receives is the one that fixes the vulnerability, which we intend to validate in runtime.

3.3.3 Program Validator

This module is responsible for two main tasks in the validation process done in two distinct phases. The first is to check whether the potential vulnerabilities found by the Vulnerability Identifier are exploitable or not. The second is to check whether the fixes applied by the Vulnerability Corrector are effective or not. To better understand how this module works, we divided it into two sub-modules: Vulnerability Checker and Code Validator.

Vulnerability Checker

This sub-module performs the first validation phase, where it uses fuzzing techniques to try to exploit the potential vulnerabilities found by the Vulnerability Identifier. To proceed with this task, it fuzzes the executable file generated earlier by the Executable Generator with inputs produced by mutation. The fuzzing starts with a standard input (e.g., a string) to trigger the loop of input mutation and test it. The fuzzing process is active for a given short period of time to try to produce an input capable of exploiting the vulnerability under test in a fast way. Note that as a slice only contains a single vulnerable execution path, it is expected that its exploitation does not take much time. During this process, the successful exploits are stored to be used later, in the second validation phase.

The potential vulnerabilities not exploited during the fuzzing period are marked as possible false positives (PFP). On the other hand, the exploitable ones are marked as such. This sub-module sends the tested files to the Vulnerability Corrector to analyze them when it finishes the fuzzing task.

Code Validator

The second validation phase of checking whether the fix applied by the Vulnerability Corrector is effective is performed by this sub-module. As in the first phase, it uses fuzzing techniques to try to exploit the file, but this time it uses the previously generated exploits stored that were able to exploit the vulner-

ability under processing. Furthermore, during the fuzzing process, these exploits are mutated to try to discover new exploits that break the applied fix.

If all exploits, i.e., the stored and the new ones, fail to break the fixed code, the applied fix is validated and considered effective. After this validation, this sub-module passes the code fixed to the Program Release Generator module (see Section 3.3.5).

3.3.4 Vulnerability Corrector

This module is responsible for one essential task in the approach pipeline, the correction of vulnerabilities. It is divided into three sub-modules: Variable and Sink Matcher, Fix Generator and Fix Applier. Each sub-module is responsible for performing different tasks required in the correction process.

Variable and Sink Matcher

This process starts by parsing the received file to locate the sensitive sinks associated with the vulnerabilities to be fixed, identify the variables used and register the sizes associated with them. At the end of the Variable and Sink Matcher execution, this collected information is passed to the Fix Generator to support the generation of the fixes for the determined cases in that correction will occur. The sensitive sinks related to buffer overflows that we handle are documented by family in Table 3.2 (columns one and two). In Section 3.4, we present a more detailed discussion about these functions.

After collecting this information, the Variable and Sink Matcher checks if the variable sizes are in accordance with their use in the sensitive sinks. There are four possible scenarios in this situation and actions to take, depicted in Table 3.1 and explained next.

When the variable size is in accordance, and the file was signalized as a possible false positive (PFP), it will be marked as a real false positive (RFP) and reported as such, and no correction is made. On the other hand, if the file was flagged as exploitable, the file is corrected by prevention. This case can happen if there is a problem with variable size checking, so the file is corrected to avoid the possible production of false negatives (FN).

When the variable size is not in accordance, and the file has been flagged as a PFP, it will be corrected on a precautionary basis because the period given for fuzzing may not have been enough to generate an exploit. On the other side, if the file has been marked as exploitable, it is considered that the file contains a real vulnerability, and it goes to the correction process.

Table 3.1: Variable and Sink Matcher possible outputs.

		File signalized as	
		PFP	**Exploitable**
Variable size in accordance	**Yes**	RFP reported No correction	Prevention Correction
	No	Prevention Correction	Vulnerable Correction

Table 3.2: List of target sensitive sinks related to buffer overflow vulnerability.

Family	Insecure Functions	Safer Functions	Correction
Input gets	gets(char *buffer);	fgets(char *buffer, int num, FILE *stream);	Use the safer function with the size of the target buffer
Input scanf family	scanf(const char *format, ...);	–	Insert the maximum number of characters to be read in the format string, which corresponds to the dimension of the target buffer minus one
	sscanf(const char *s, const char *format, ...);		
	fscanf(FILE *stream, const char *format, ...);		
	vscanf(const char *format, va_list arg);		
	vsscanf(const char *s, const char *format, va_list arg);		
	vfscanf(FILE *stream, const char *format, va_list arg);		
Output	sprintf(char *s, const char *format, ...);	snprintf(char *s, size_t n, const char *format, ...);	Use the safer function with the maximum number of characters to be written, according to the capacity of the target buffer
	vsprintf(char *s, const char *format, va_list arg);	vsnprintf(char *s, size_t n, const char *format, va_list arg);	
String copy	strcpy(char *dst, const char *src);	strncpy(char *dst, const char *src, size_t n);	Use the safer version with the number of characters to be copied, according to the capacity of the target buffer
String concatenation	strcat(char *dst, const char* src);	strncat(char *dst, const char *src, size_t n);	Use the safer version with the number of characters to be concatenated, according to the available capacity of the target buffer

Fix Generator

When the file enters to the correction process, the Fix Generator sub-module analyzes the sensitive sinks' instructions to understand what type of correction should be applied in that case. These sensitive sinks instructions correspond to the functions documented in Table 3.2 (column two). Each sensitive sink works differently and can be used differently, so several correction possibilities exist. However, the fixes generated by the Fix Generator are based on the analysis of the functions described in Section 3.4.

It is necessary to analyze the sinks used and correctly calculate the sizes of the variables they use to generate the fixes. This analysis and the subsequent size calculation are based on the information received from the Variable and Sink Matcher module. In addition, the fixes mentioned are generated with the help of templates containing the instructions that correspond to the safe uses of the sensitive sinks with generic parameters. For each case, these generic parameters are modified by those specific to that case.

In Table 3.2 we can observe the safe version of the functions used in the corrections and a summary of the fixes applied for each case (columns three and four).

Finally, after generating all the fixes for the sensitive sinks present in the file, this sub-module makes a mapping between the generated fixes and the line of code where they should be applied in the file. This mapping is then sent to the Fix Applier, as well as the vulnerable file and the fix to be applied.

Fix Applier

Using the mapping received from the Fix Generator, the Fix Applier identifies which lines should be modified and what changes should be made to those lines. It reads the lines from the file to be fixed and copies them to a new file. The lines that are present in the mapping, before being copied to the new file, are modified according to the fix received. This way it is created a new file with the applied fixes that is next transmitted to the File Compiler to be compiled and generate a new executable file.

3.3.5 Program Release Generator

When the second phase of the validation process (Section 3.3.3) ends without any exploit (the previously generated and new ones) breaking the fixes, and it is found that the applied fix does not spoil the program's functioning, it means that the fix is effective and can be used to correct the original program. The Program Release Generator module is responsible for performing this correction, i.e., for integrating the correction produced and validated in the original files.

It receives the original program files, the corrected slices containing the generated fixes, and the original line numbers associated with the slices. By mapping the lines between the original program and the slices, it inserts the fixes in the respective places in the original program, thus resulting the final program with the applied fixes, which will be a new and corrected version of the original program.

Listing 3.4 shows the example file with the applied correction that results from the execution of this module. As we can observe, in line 6 of the original file (Listing 3.1) the `"%s"` argument of `scanf` was corrected to `"%14s"` to only be read fourteen characters to the buffer variable since this last can only store fifteen. We recall that the fifteenth position of the buffer is reserved for the null character (`"\0"`), and hence we can only occupy the first 14 positions with data. In this way, the existing buffer overflow is removed and the code correctly fixed.

3.4 Sensitive Sinks

In this section, we discuss the functions presented in Table 3.2, which correspond to the sensitive sinks we want to address, and how these should be used.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buffer[15];
    printf("Enter a string\n");
    scanf("%14s", buffer);
    printf("Entered string: %s\n", buffer);
    return(0);
}
```

Listing 3.4: Example file fixed.

3.4.1 Input `gets`

The `gets` function is very hard to use safely. It has only one parameter, a pointer to a buffer. It reads a line from the standard input and stores it in the target buffer without checking its size. The problem is that it is impossible to know, in advance, the amount of data that the function will receive if there are no restrictions. For this reason, it is recommended to use the `fgets` function, which has another parameter that specifies the maximum number of characters to be read. This function reads at least one less character than the size specified to leave space for the terminating null character, which is appended automatically after the copied characters.

The fix for the `gets` function consists of replacing its instruction with a similar one using the `fgets` function and specifying the maximum number of characters to read, corresponding to the size of the target buffer variable. The size is calculated by the Fix Generator and applied to the template corresponding to the `fgets` function, which is modified so that the arguments match those received in the original instruction. In this way, the new instruction that will replace the previous one is generated with the correct size and arguments.

Listing 3.5 shows an example of how to use the `fgets` function. In the example, the function has specified 15 characters, but the function will read 14 characters and fill the last one with the terminating null character. Therefore, there is no need to add it as the last character, but it is necessary to consider this when passing the right size in the parameter. In the example, the red instruction corresponds to the insecure and problematic sensitive sink we want to address, while the green line is the instruction that will replace the red one, i.e., the fix we will apply.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buffer[15];
    gets(buffer);
    fgets(buffer, 15, stdin);
    printf("Buffer Content: %s\n", buffer);
    return(0);
}
```

Listing 3.5: Usage of function `fgets`.

3.4.2 Output functions

The functions `sprintf` and `vsprintf` send formatted strings as output to a buffer. They take an argument, a format string that contains the text to write to the buffer. It can contain conversion specifications that are replaced by the values specified and formatted as requested. For instance, a format string containing "%s" is replaced by a string. Its size must be considered when writing to the buffer. One way to do this is to specify a precision with a string conversion specification (e.g., ".5%s"). This way, only the first characters of the string up to the given value are considered (in the example, the first 5 characters). Even using the precision specification, it is necessary to be careful with the limit placed because it is essential to consider the size of the target buffer and, in particular, the terminating null character. Format strings may have different conversion specifications, which makes them difficult to use and favors the appearance of errors if the sizes are not well calculated. Therefore, it is necessary to be very careful when using the specifications and calculating the buffer size.

Functions `snprintf` and `vsnprintf` are equivalent to `sprintf` and `vsprintf`, respectively, but they have an extra argument – n – which specifies the maximum number of characters that may write to the buffer, including the terminating null character. If n is too small to accommodate the complete output string, then the function writes only the first n-1 characters of the output, followed by a null character, and discards the rest. Because of this, they are considered less dangerous and easier to use.

When the `sprintf` and `vsprintf` functions are identified as vulnerable, the fix is to modify the instruction that corresponds to them by an equivalent one but using the `snprintf` and `vsnprintf` versions, respectively. Also, it is needed to determine the n value that is passed in the parameter corresponding to the maximum number of characters that could be written to the buffer. After the value is calculated, it is applied to the template for these functions, and the new instruction is generated with the fix applied.

Listing 3.6 shows an example of a program vulnerable to buffer overflow in the red line 6. The `sprintf` function is used without checking the number of characters copied. In the green line 6, the solution generated by our approach using the `snprintf` function is shown.

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     char buffer[15];
5     char *str = "Buffer Overflow String";
6     sprintf(buffer, "%s", str);
6     snprintf(buffer, sizeof(buffer), "%s", str);
7     printf("Buffer Content: %s\n", buffer);
8     return(0);
9 }
```

Listing 3.6: Example of `sprintf` correction.

3.4.3 Input `scanf` family

The functions `scanf`, `sscanf`, `fscanf`, `vfscanf`, `vsscanf`, like the previous ones, also receive a format string as a parameter. However, in these functions, the format string indicates how to read and interpret the input data, to place in some additional arguments that the function may expect. The number

of these arguments should be the same as the number of tags specified in the format parameter. Once again, it is necessary to be careful with the size of the buffers that will receive the input data. Since the size of the input is usually not known before being read, it is always better to put the size of the destination buffer in the format string. The functions of this family, such as `fgets` function, automatically place the terminating null character.

The process to correct the `scanf` family functions is more complex because there is no safe version of these functions, and it is not possible to add another parameter containing the target buffer size.

The only way to correct the use of these functions is to modify the format string to take into account the size of the target buffer. To generate the instruction correctly, it is necessary to know exactly the size of the buffer used, i.e., it is not possible to use expressions such as `sizeof(buffer)` or `strlen(buffer)`, inside the format string. This peculiarity makes it very difficult to correct the cases where it is not possible to know the exact buffer size through static analysis, namely when a buffer is dynamically allocated, and the size depends on the result of function execution. The solution to correct this case is to create a variable that stores the result of the function execution. This variable is then used to make the allocation and will be used to generate a new format string with the correct buffer size. In this case, it will be the variable's value minus one to leave space for the terminating null character.

Another complicated situation is when a buffer is passed as a parameter to a function and used within that function. To correct such cases, the function declaration and call are modified to receive an additional parameter that corresponds to the size of the buffer that is passed. This new parameter with the size is then used to generate a new format string that contains the correct value. In this case, it will be the value passed minus one to leave space for the terminating null character. The old format string is replaced by the new one in the function, and, thus, a new corrected instruction is generated.

In cases where it is possible to determine the exact buffer size through static analysis, the correction consists of modifying the format string present in the instruction by adding the correct number of characters to be read, counting the space for the terminating null character.

Listing 3.7 shows a way how to use correctly the `scanf` function. As we can notice, the buffer size is 10, but in the format string in the second line 5, the maximum number of characters that can be read is 9 because it is necessary to leave space for the terminating null character.

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     char buffer[10];
5     scanf("%s", buffer);
5     scanf("%9s", buffer);
6     printf("Buffer Content: %s\n", buffer);
7     return(0);
8 }
```

Listing 3.7: Usage of the `scanf` function with a wrong format string and the respective correction.

3.4.4 String Copy

The `strcpy` function has two parameters, a destination buffer and a string to be copied to that buffer. When using this function, care must be taken with the size of the parameters. It is necessary to verify the

string size to check if it fits in the buffer. Besides checking the size, an alternative to solve this problem can be to allocate enough memory to the buffer dynamically. Another way is to use the `strncpy` function, which has one more parameter that receives the number of characters to be copied. Although this function is considered safe, care must be taken because there are some situations where problems may exist. There may be a risk that the string will not have the terminating null character if it is longer than the buffer. Such a situation can cause problems in program execution because there are functions whose correct operation depends on the presence of the terminating null character (e.g., `strlen` function). When using the `strncpy` function, you must put the terminating null character in the last position of the buffer.

The correction adopted for the `strcpy` function is to use the `strncpy` version instead. The Fix Generator needs to determine the correct number of characters that can be copied into the target buffer to use this function correctly, with the help of the information it receives. After doing the calculation, it uses the appropriate template for this function and modifies it to consider the value calculated.

In addition to modifying the `strcpy` function statement by the new `strncpy` function statement, it inserts a new statement that adds the terminating null character to the last position of the target buffer. It uses the size calculated before and a template specific to this instruction to generate this statement.

The `strcpy` function could be corrected in another way by adding an `if` statement with the variable size check before the `strcpy` instruction. However, we chose the first correction because, in this way, no more execution paths are generated than the ones that already existed, which keeps the file as original as possible.

Listing 3.8 shows an example of the correct use of the `strncpy` function. Note that in line 8, it is ensured that the last position of the buffer contains the terminating null character. The first line 7 corresponds to the incorrect usage of the `strcpy` function. The second line 7 and line 8 correspond to the correction generated for this case.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[]) {
5      char buffer[10];
6      char *str = "Example string";
7      strcpy(buffer, str);
7      strncpy(buffer, str, sizeof(buffer) - 1);
8      buffer[sizeof(buffer) - 1] = '\0';
9      printf("Content of buffer: %s\n", buffer);
10     return(0);
11 }
```

Listing 3.8: Example of `strcpy` fix using the `strncpy` function.

3.4.5 String Concatenation

The `strcat` function is similar to the `strcpy` function. However, the string to be copied is appended to what is in the destination buffer. In this case, it is necessary to check if the buffer is big enough to contain what already has plus the string that will be added. This function also has a secure version, `strncat`, which unlike `strncpy`, adds the null character at the end of the buffer. In this case, the

concern in its use is the size that is passed as a parameter. This one must contemplate the size of what already exists in the buffer, plus what will be added, and the space for the null character.

As in the previous case, the correction adopted for the `strcat` function uses the `strncat` version for the same reasons mentioned above. Again, it is necessary to calculate the correct number of characters that can be copied, considering the variable's sizes, their content, and the space for the terminating null character. Once again, the value is used jointly with the template resulting in the new instruction to substitute the old one.

In Listing 3.9 we can see an example program where the first line 7 is vulnerable to buffer overflow due to incorrect usage of the `strcat` function. The second line 7 represents the fix generated by our approach for this case using the `strncat` function. When executing the program with the first line 7 an error appears, and the program crashes. On the other side, with the second line 7, the instruction in line 8 would print the result `Hello wor`, because only three characters are copied from the `src` buffer. This happens because the `dst` buffer is of size ten and already contains six characters, leaving space for four characters, but is necessary space for the terminating null character, so only three characters are copied.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[]) {
5      char src[10] = "World";
6      char dst[10] = "Hello ";
7      strcat(dst, src);
7      strncat(dst, src, sizeof(dst)-strlen(dst)-1);
8      printf("Destination string : %s", dst);
9      return(0);
10 }
```

Listing 3.9: Incorrect usage of the `strcat` function and the respective correction.

Chapter 4

Implementation

This chapter describes the implementation of the proposed solution that supports the approach presented in Section 3.3. The prototype developed was implemented in Python, and some tools were used to facilitate the realization of some specific phases of the tool pipeline. Section 4.1 describes the tools used to assist the implementation of our solution. Section 4.2 presents the main algorithms of the tool and explains how they work.

4.1 Tools used

This section describes the tools used in the implementation of some tasks performed in the pipeline of our solution.

4.1.1 Flawfinder

Flawfinder [7] is a static analysis tool that scans C/C++ source code and reports potential security flaws. It works by using a built-in database of C/C++ sensitive functions with well-known problems, such as the `strcpy` function associated with buffer overflow risks, and then takes the source code for matching it against the name of those functions. In such match case, it hits the instructions of the source code as potential flaws, i.e., vulnerabilities. These instructions contain some sensitive functions from its database. In the end, it produces a list of hits, i.e., a list of potential security flaws, sorted by risk. It was chosen to help detect vulnerabilities because the functions we want to address are present in its database of vulnerable functions. Therefore, Flawfinder is used in our Vulnerability Identifier module to detect buffer overflows. In addition, it is a simple tool that is easy to use, install, and integrate.

4.1.2 American Fuzzy Lop (AFL)

American Fuzzy Lop (AFL) [1] is a fuzzer for C/C++ programs to discover vulnerabilities they have, producing the test cases that exploit them. It mutates an input seed given at the start of fuzzing to generate new test cases, which could lead to the discovery of new execution paths of the program under test. It interacts with the target program's binary while processing the input passed and monitors what segment of code was triggered, i.e., it keeps track of code paths being triggered. Based on these paths, it mutates the seed files to discover new execution paths. Simultaneously, it checks if the test cases are able to exploit some existent vulnerability in those paths. This tool was chosen to be used in the

Program Validator module to confirm the existence of the potential vulnerabilities and to validate the fixes generated by our approach due to its ability to detect vulnerabilities derived from the clever way it modifies seeds.

4.1.3 Pycparser

Pycparser [14] is a parser for the C language, written in Python. It is a module designed to be easily integrated into applications that need to parse C source code. Pycparser aims to support the full C99 language, and some features from C11 are also supported. It parses C code into an Abstract Syntax Tree (AST). This AST contains nodes built by Pycparser that correspond to the different constituents of the program (e.g., variable declarations, function calls). Pycparser was chosen as the parser to be used in different modules of our solution, namely the Vulnerability Identifier, the Executable Generator, and the Vulnerability Corrector, in which it is necessary to parse the code under analysis.

4.2 Algorithms

This section presents the algorithms in pseudo-code that describe the implementation of the main functions of the tool and explains how they work.

4.2.1 Main algorithm

```
   Input: Path to C/C++ program files
   Output: Fixed Program
1  Function Main(path):
2      corrections ← [];
3      origLines ← [];
4      slices ← identifyVulnerabilities(path);
5      for slice in slices do
6          file, lines ← generateFile(slice, path);
7          origLines.append(lines);
8          executable ← createExecutable(file);
9          exploits, hasExploit ← checkVulnerabilities(executable, seed);
10         fileFixed, fixes ← correctVulnerabilities(file, hasExploit);
11         if fileFixed then
12             newExecutable ← createExecutable(fileFixed);
13             exploits, hasExploit ← checkVulnerabilities(newExecutable,
                exploits);
14             if not hasExploit then
15                 corrections.append(fixes);
16             end
17         end
18     end
19     if corrections then
20         generateProgramRelease(path, corrections, origLines);
21     end
22 End
```

Algorithm 1: Main algorithm.

Algorithm 1 describes the highest level steps performed by the tool, i.e., the main algorithm. It starts by initializing two lists, `corrections` where the generated fixes will be saved and `origLines` that

will store the mapping between the original lines of the program under test and the corresponding lines in the new files generated for the respective slices. Next, it calls the `identifyVulnerabilities` function, which runs Flawfinder on the received program to identify potential vulnerabilities and then generates slices of code for the potential vulnerabilities found (see Algorithm 2).

Once the slices are generated, for each one of them, the function `generateFile` is called to create a file that contains that slice and all the information needed for the file to be compilable (see Algorithm 3). This step not only creates the new file but also creates the mapping between its lines and the lines of the original file, which is appended to the `origLines` list. Afterwards, the `createExecutable` function is called, which compiles the file generated using the AFL compiler to instrumentalize the program so that the generated binary can be executed by the AFL. After the binary has been generated, the `checkVulnerabilities` function is called, which runs AFL to fuzz the previously generated binary. After this execution, the function returns the exploits generated by AFL and the flag that identifies the file as a possible false positive or exploitable (see Algorithm 4). Subsequently, the file and the flag are passed to the `correctVulnerabilities` function, which analyzes the file and checks if it needs to be corrected. If the file needs to be corrected, the necessary fixes are generated and applied to the code, and a new file is created with the new corrected code (see Algorithm 5). If a new file is generated, then that file is compiled in the way explained above, again using the `createExecutable` function to generate a new binary. This new binary is then passed to the `checkVulnerabilities` function along with the exploits generated in the previous run of this function. This time the function runs AFL to fuzz the new binary and returns some possible exploits and the `hasExploit` flag resulting from fuzzing it. This flag is checked, and if it has the value False, then it means that no exploits were found, and therefore, the previously generated fixes have been validated and are added to the `corrections` list.

After iterating over all slices, a validation is made to check if the `corrections` list is empty. If it is not, it means that we have a list with all the validated fixes and another list with the mapping of the original lines and the lines where fixes were generated. Finally, these lists are passed along with the original program to the `generateProgramRelease` function, which applies the fixes to the original code and creates new files with these modifications, resulting in a new fixed program (see Algorithm 6).

4.2.2 Vulnerability Identifier algorithm

Algorithm 2 is responsible for scanning the received program for potential vulnerabilities. It starts by initializing an empty list `slices`, where the slices of each possible vulnerability found will be stored. Next, the function `runFlawfinder` is called, which runs Flawfinder on the received program files, which results in a list of hits of all the weaknesses found, with information about the line and the file they are. This list is filtered to contain only the hits related to the sensitive sinks we want to address. Next, the `getFileLines` function is called, which goes through the filtered list to identify the files and lines of the hits, then reads those files and copies the respective lines to a list that will contain all the lines of the hits found and their file. For each line in this list, the `parseLine` function is called, which uses the Pycparser to parse the line to identify and store the variables present in the line.

Afterwards, the `parseFile` function is called, which uses the parser to generate the AST of the file where the line under analysis is located. The AST is processed to identify the variable statements

```
Input: Path to C/C++ program files
Output: List of code slices
1  Function identifyVulnerabilities(path):
2      slices ← [];
3      hits ← runFlawfinder(path);
4      lines ← getFileLines(hits, path);
5      for line in lines do
6          variables ← parseLine(line);
7          slice ← parseFile(path, variables, line);
8          slices.append(slice);
9      end
10     return slices;
11 End
```

Algorithm 2: Algorithm performed by the Vulnerability Identifier.

identified earlier and these statements are stored in a list together with the line. This junction forms a slice that is added to the `slices` list with the identification of which file that slice belongs to. Finally, after all the lines have been analyzed, the `slices` list is returned containing the slices of all the hits.

4.2.3 File Generator algorithm

```
Input: Code slice; Path to C/C++ program files
Output: File with the code slice; Mapping of the file's lines
1  Function generateFile(slice, path):
2      variables, sinks ← parseSlice(slice);
3      instructions ← parseFile(path, variables, sinks);
4      file, mapLines ← createFile(slice, instructions);
5      return file, mapLines;
6  End
```

Algorithm 3: Algorithm performed by the File Generator.

Algorithm 3 is applied by the File Generator and corresponds to the operations performed to generate a compilable file containing the slice received. It starts by calling the `parseSlice` function, which uses Pycparser to generate the AST representing the slice. The AST is processed to identify the variables and sensitive sinks used in the slice. Subsequently, the `parseFile` function is called, which uses the parser to generate the AST of the file to which the slice is associated. It then analyzes this AST in a bottom-up approach by processing the instructions to identify those that use or depend on the variables and sinks used in the slice. In this way, it is possible to identify the instructions that need to be added into the file to have a complete execution path and to the file can be compiled. These instructions include the directives needed to compile the code and other variables or function calls/declarations needed, as well as the line where they are in the file.

After determining which additional instructions are needed, the `createFile` function is called, which opens a new file and writes to it the directives, the function declarations determined previously, and, finally, the `main` function containing the remaining instructions and the slice. Each time this function writes an instruction to the new file, it saves in a list the correspondence between the line number of the instruction in the original file and the value corresponding to the line number of the instruction in the

new file. This value is obtained from a counter incremented when a new line is written to the file. This is how the mapping between the line numbers of the original file and the new file is done. The execution of this function results in creating a compilable and executable file containing the slice with the potential vulnerability and a mapping between the line numbers of the original file and the new file.

4.2.4 Program Validator algorithm

Input: Executable file; Input seeds
Output: Generated exploits; Has exploit tag

```
1 Function checkVulnerabilities(executable, seeds):
2     hasExploit = False;
3     exploits ← runAflFuz(executable, seeds);
4     if exploits then
5         hasExploit = True;
6     end
7     return exploits, hasExploit;
8 End
```

Algorithm 4: Algorithm performed by the Program Validator.

Algorithm 4 describes how the Program Validator works in checking potential vulnerabilities and validating the fixes generated by the Vulnerability Corrector. It starts by initializing a boolean variable to false that serves as a flag to indicate whether the file is exploitable or not. Next, the function `runAflFuz` is called, which runs a subprocess where AFL runs over the binary generated from the file under test for a given period of time. Some seeds are passed in as input for AFL to start executing and making mutations from those seeds. When the subprocess execution time is over, AFL is stopped, and the folder where the generated exploits are stored is analyzed to verify if any exploits were generated or not. If there are exploits, they are stored in a list. Otherwise, it remains empty. Afterwards, it is verified if the list has exploits. In the affirmative case, the previously initialized flag becomes true. Finally, the list of exploits and the flag are returned. This algorithm works the same way in both Program Validator execution phases. However, in the second phase, i.e., in the validation of the fixes, the seeds given as input contain not only the seed passed in the first phase but also the exploits generated in that phase.

4.2.5 Vulnerability Corrector algorithm

Algorithm 5 represents the general execution of the Vulnerability Corrector module. It starts by initializing the list `fixes`, where will be stored the generated fixes with the indication of the file they belong to and the lines where they should be inserted. Next, the `parseFile` function is called, which uses Pycparser to generate the AST of the received file, which is processed to identify which variables exist and store the relevant information about them, namely their names and sizes. In addition to the variables, function calls are also processed to identify the desired sensitive sinks and check which variables they use and how.

After collecting this information, the `checkFunctions` function is called to analyze the usage of the variables in the respective sensitive sink. This analysis consists in verifying if the sink correctly uses the variables considering their sizes, i.e., verify if there is a correspondence between the way the variable was used and the size of that variable respecting the aspects discussed in Section 3.4. If this

```
   Input: File to correct; Has exploit tag
   Output: Fixed file; List of fixes
1  Function correctVulnerabilities(file, hasExploit):
2      fixes ← [];
3      variables, sinks ← parseFile(file);
4      hasMatch ← checkFunctions(variables, sinks);
5      if hasMatch and not hasExploit then
6          return fixes;
7      else
8          for sink in sinks do
9              fix ← createFix(variables, sink, templates);
10             fixes.append(fix);
11         end
12     end
13     fileFixed ← applyFixes(file, fixes);
14     return fileFixed, fixes;
15 End
```

Algorithm 5: Algorithm performed by the Vulnerability Corrector.

correspondence exists, it returns the variable `hasMatch` with the value true and otherwise with the value false. If the variable `hasMatch` is true and the variable `hasExploit` is false, it means that no real vulnerabilities were identified. In that case, the execution ends with the return of the `fixes` list that is empty because no fixes were generated. Otherwise, the function enters a loop for each sensitive sink in the code, calling the `createFix` function, which receives the previously collected information and the templates of the fixes to apply for each sink.

These templates are strings that represent the correct usage of the respective sensitive sink with default parameters. The `createFix` function identifies the sink, the variables used and the template corresponding to that sink. Then calculates the sizes that should be used and replaces the default parameters in the template with the variables identified and the right sizes. This process generates the correction that consists of a new instruction with the correct usage of the identified sensitive sink. After the correction is generated, it is added to the `fixes` list with the file and line where it should be inserted. When this process is finished for all the sensitive sinks identified, the `applyFixes` function is called and receives the file and the list of fixes. This function reads all the file's lines and stores them in a list. It then iterates over the list of fixes and takes the line number where the fix should be inserted. Next, it modifies the position in the list of lines corresponding to that line with the proper fix. At the end of the iteration, the list of lines contains all the corrections generated. Afterwards, the file is rewritten with these new lines, resulting in a new file with all the corrections applied. Finally, this new file and the list of fixes are returned.

4.2.6 Program Release Generator algorithm

Algorithm 6 shows the procedures performed by the Program Release Generator to correct the program initially received. It starts by iterating over the received files and calls the `readFile` function, which opens the file and reads all its lines into a list. It then iterates over the `fixes` list and the `origLines` list corresponding to that file to find the line from the original file that matches the line from the file generated during testing. When it finds that match, it inserts the fix into the given index of the `lines`

```
Input: Path to C/C++ program files; Fixes generated; Original lines
Output: Fixed Program
1  Function generateProgramRelease(path,fixes,origLines):
2      for file in path do
3          lines ← readFile(file);
4          for fix in fixes[file] do
5              for origLine in origLines[file] do
6                  if fix.line == origLine.new then
7                      lines[origLine.old] = fix.correction;
8                      break;
9                  end
10             end
11         end
12         writeLines(lines);
13     end
14 End
```

Algorithm 6: Algorithm performed by the Program Release Generator.

list. At the end of the file iteration, it calls the function `writeLines`, which opens a new file and writes the `lines` list into it. This process is repeated for all files, thus generating a new version of each one containing all the fixes generated by the tool.

Chapter 5

Evaluation

This chapter describes the evaluation process performed to evaluate our tool and discusses the results obtained. Section 5.1 describes the evaluation setup and identifies the metrics used for the evaluation. Sections 5.2 and 5.3 present and discuss the evaluation with, respectively, a synthetic dataset and real applications, both comprising C programs.

To evaluate the tool, it is necessary to reason about the challenges identified in Section 3.1 and the solutions we proposed to solve them. Based on that information, there are four main aspects that we should evaluate in the tool: its ability to find vulnerabilities, its ability to build compilable and executable files, its ability to correct vulnerabilities, and the effectiveness of the generated fixes. Considering these aspects, we defined the following questions:

Q1 Is the tool capable of detecting potential vulnerabilities associated with the functions addressed?

Q2 Can the tool create correct code slices for the potential vulnerabilities found?

Q3 Is the tool capable of generating compilable and executable files for the slices created?

Q4 Can the tool confirm that the potential vulnerabilities found are really exploitable?

Q5 Is the tool capable of correcting the vulnerabilities?

Q6 Are the fixes generated by the tool effective?

Q7 Is the tool capable of processing real applications and fix vulnerabilities?

With the intent of answering these questions, the evaluation process was conducted as described in the upcoming sections.

5.1 Evaluation Setup and Metrics

To evaluate the tool's capabilities more thoroughly, we divided the evaluation into two parts. Firstly, we used a synthetic dataset of test cases (small C programs) taken from Software Assurance Reference Dataset (SARD) [13] to evaluate the tool's performance and validate its capabilities.

The test cases that are part of the dataset were previously classified as vulnerable or not vulnerable, serving as the ground truth for comparison with the results obtained by the tool. With this data, it is

Table 5.1: General confusion matrix.

		Tool classification		Total
		Vulnerable	**Not Vulnerable**	
Ground Truth	**Vulnerable**	TP	FN	$TP + FN$
	Not Vulnerable	FP	TN	$FP + TN$
	Total	$TP + FP$	$FN + TN$	$TP + FN + FP + TN$

possible to create a confusion matrix like the one presented in Table 5.1, from which we calculate the values of True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN).

Once these values are determined, it is possible to calculate some evaluation metrics to have statistical data to assess the tool's performance. Next, we describe the metrics we decided to use, their meaning in the context of our evaluation, and the respective formulas from which we can calculate their values.

Accuracy (ACC): Measures the percentage of correct decisions made by the tool.

$$ACC = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.1}$$

False Negative Rate (FNR): Measures the percentage of vulnerable test cases missed by the tool.

$$FNR = \frac{FN}{FN + TP} \tag{5.2}$$

False Positive Rate (FPR): Measures the percentage of not vulnerable test cases incorrectly identified as vulnerable by the tool.

$$FPR = \frac{FP}{FP + TN} \tag{5.3}$$

Precision (Pr): Measures the percentage of vulnerable test cases correctly identified by the tool.

$$Pr = \frac{TP}{TP + FP} \tag{5.4}$$

Recall or True Positive Rate (TPR): Measures the percentage of vulnerable test cases the tool identified as such.

$$TPR = \frac{TP}{TP + FN} \tag{5.5}$$

Specificity or True Negative Rate (TNR): Measures the percentage of not vulnerable test cases that the tool identified as such.

$$TNR = \frac{TN}{TN + FP} \tag{5.6}$$

F-Score: It is the harmonic mean of precision and recall.

$$F - Score = 2 * \frac{Pr * TPR}{Pr + TPR} \tag{5.7}$$

Finally, in the second phase, we used some applications written in C taken from SourceForge repository and from a XIVT's partner to test the tool's capabilities to process real programs. At this stage, the metrics presented above were not calculated because there is no classification of the application files to compare with the results obtained by the tool.

5.2 Evaluation with SARD dataset

SARD is a dataset maintained by the National Institute of Standards and Technology (NIST). It contains several thousand test cases for different types of vulnerabilities in a wide variety of programming languages. Since our work is focused on C, we are only interested in test cases for that programming language.

We gathered a total of 1075 test cases from SARD. All these test cases contain some of the functions that we want to address and were manually analyzed and classified as vulnerable or not vulnerable. We classified 560 as vulnerable and 515 as not vulnerable. Table 5.2 summarizes the number of test cases collected for each function type.

Table 5.2: Summary of test cases collected from SARD.

	Function	Total test cases
Input	gets	33
	scanf	120
	sscanf	120
	fscanf	120
	vscanf	30
	vsscanf	30
	vfscanf	30
Output	sprintf	56
	vsprintf	30
String copy	strcpy	115
String concatenation	strcat	115
Multiple Functions		276
Total		1075

During the evaluation, the tool processed all test cases and generated all slices and executable files correctly. Based on the classification we made manually and the results obtained with the tool, we built the Table 5.3, representing the confusion matrix of the Vulnerability Identifier, Program Validator, and Vulnerability Corrector modules. Using the values from this table, we calculated the metrics defined in Section 5.1 for each module. Table 5.4 shows these metrics, from which it is possible to visualize the evolution throughout the tool's pipeline. Note that the Program Validator values shown in the table are relative to the first execution of this module in the tool's pipeline, i.e., the confirmation of vulnerability existence.

Table 5.3: Confusion matrix of the modules evaluated.

		Tool Classification					
		Vulnerability Identifier		Program Validator		Vulnerability Corrector	
		Vulnerable	Not Vulnerable	Vulnerable	Not Vulnerable	Vulnerable	Not Vulnerable
Ground Truth	Vulnerable	560	0	560	0	560	0
	Not Vulnerable	515	0	0	515	30	485
	Total	1075	0	560	515	590	485

Table 5.4: Summary of the calculated evaluation metrics.

Metric	Vulnerability Identifier	Program Validator	Vulnerability Corrector
Accuracy (ACC)	0.52	1.00	0.97
False Negative Rate (FNR)	0.00	0.00	0.00
False Positive Rate (FPR)	1.00	0.00	0.06
Precision (Pr)	0.52	1.00	0.95
Recall or True Positive Rate (TPR)	1.00	1.00	1.00
Specificity or True Negative Rate (TNR)	0.00	1.00	0.94
F-Score	0.69	1.00	0.97

Based on the data in Table 5.3, we can see that the Vulnerability Identifier module identified all test cases as vulnerable. Since all test cases have a function that we want to address, then this shows that the module can identify these functions. Furthermore, by analyzing the recall value of this module stated in Table 5.4, we can observe that all the vulnerable files were identified as such. Therefore, based on these two results, we can answer affirmatively to question **Q1**.

Regarding the Executable Generator module, we can conclude that it fulfilled its role. It was able to generate compilable and executable files correctly. An indicator of this result is that the tool was able to process all test cases. All modules processed all test cases, as we can see in Table 5.3. In addition, we can observe that the Program Validator module detected and exploited all vulnerable cases correctly, which shows that the executable files generated were correctly built and contain the slices with the vulnerabilities. Furthermore, the module has the capacity of invalidating the false positives generated by the Vulnerability Identifier. Therefore, even though this last module has the highest FPR, the Program Validator achieves null FNR and FPR, i.e., a precision, recall, and F-Score equal to 100 %. This answers questions **Q2** and **Q3** since the two are related.

The results obtained by the Program Validator depicted in both tables show that this module was able to correctly identify all vulnerable test cases, as well as the not vulnerable test cases. These results serve to answer the **Q4** question since this module detected all the real vulnerabilities.

As mentioned before, the results of the Program Validator shown in these tables are relative to the first execution of this module in the tool's pipeline. In the second execution of this module, all test cases were classified as not vulnerable since the module did not find any problems during the execution. This result indicates that the Vulnerability Corrector module was able to generate correct syntactically and effective corrections as they removed the existing vulnerabilities. With this result, we can answer question **Q5** because the tool was able to generate the right corrections and question **Q6**. After all, the generated corrections proved to be effective.

It is possible to observe in both tables that the Vulnerability Corrector module presented some false positives, i.e., it corrected some test cases that were not vulnerable. The reasons for the existence of false positives are discussed next.

One of the reasons for the existence of false positives is due to a limitation of static analysis. Some test cases contain dynamic arrays whose size is calculated based on the result returned by executing some function. In that cases, it is not possible to statically determine the size of the array. Therefore, we decided that in this situation, a correction would always be generated, for prevention, to avoid false negatives. Listing 5.1 shows an example of this type of situation. In this example, the function `fgets`

reads some text provided by the user and writes it into the array `userstr` (line 8). Then, the length of this text is determined and stored in the integer variable `size` plus one character (line 9). Using this value, the program allocates memory to the char pointer `userstr_copy`, to which the content of the array `userstr` is copied (lines 10 and 11). In this example, there is no buffer overflow because the `userstr_copy` was created with enough size to store the contents of the array `userstr`. However, through static analysis, it is not possible to determine the `size` value because it is calculated at runtime. Therefore, it is not possible to compare the `strcpy` function parameters to check whether or not there is a buffer overflow, which leads the Vulnerability Corrector module to fix this code.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #define MAXSIZE 40
5
6  int main(int argc, char **argv) {
7      char userstr[MAXSIZE];
8      fgets(userstr, MAXSIZE, stdin);
9      int size = strlen(userstr) + 1;
10     char *userstr_copy = malloc(sizeof(char) * size);
11     strcpy(userstr_copy, userstr);
12     puts(userstr_copy);
13     free(userstr_copy);
14     return(0);
15 }
```

Listing 5.1: Example of the first reason for false positives.

```
1  #include <stdio.h>
2  #include <string.h>
3  #define MAXSIZE 40
4
5  char *shortstr(char *p, int n, int targ) {
6      if(n > targ)
7          return shortstr(p+1, n-1, targ);
8      return p;
9  }
10
11 void test(char *str) {
12     char buf[MAXSIZE];
13     str = shortstr(str, strlen(str), MAXSIZE-1);
14     strcpy(buf, str);
15     printf("result: %s\n", buf);
16 }
17
18 int main(int argc, char **argv) {
19     char userstr[100];
20     fgets(userstr, sizeof(userstr), stdin);
21     test(userstr);
22     return(0);
23 }
```

Listing 5.2: Example of the second reason for false positives.

Another situation that results in false positives for the Vulnerability Corrector is when some string is truncated by some function before being used in the `strcpy` or `strcat` functions. Through static

analysis, it would be very difficult to calculate the length of the string after going through a function that modifies its size, so we decided that in these situations, a correction would also be generated, once again for prevention. Listing 5.2 depicts a program in which this situation occurs. The program reads a string entered by the user, which is copied into the array `userstr` and can have a size up to 100 bytes. This string is copied in the `test` function into the `buf` array which has size of 40 bytes. In case the user-entered string is bigger than 40 bytes, there would be a buffer overflow. However, the string is not copied directly; it is used in the `shortstr` function, which will truncate the string to a size less than 40 bytes so that it can be copied without problems. So this program has no buffer overflow.

Overall, the Vulnerability Corrector had a recall of 100 % and a precision of 95 %.

5.3 Evaluation with Real Applications

SARD dataset allowed the evaluation of the tool's capability to deal with all the functions addressed and to measure its performance. However, SARD's test cases might not represent real applications accurately. Therefore, we obtained six applications in pre-alpha and beta versions from SourceForge to test our tool with real code. Besides these applications, we also tested a railway driver software made available by a partner of the project in which this work is inserted.

Table 5.5 presents a summary of the applications that were collected. The table includes the version of the application, the number of files it contains, the number of lines of code (LoC), and a brief description of the application. Regarding the code provided by the partner, it is a driver of a Propulsion Control Subsystem (PCS) of a railway, which contains 20 files with 5483 lines of code. In total, the tool analyzed 209 files, corresponding to 132,209 lines of code.

Table 5.5: Summary of the applications used in the evaluation.

Application	**Version**	**No. Files**	**No. LoC**	**Description**
Zervit	0.4	17	1014	Http portable server
Macgen	1.1	1	15	Random MAC address generator
sSocks	0.0.14	30	3477	Socks5 Server
Tiny HTTPd	0.1	3	765	Simple webserver
LIBPNG	1.6.37	88	57075	Library for supporting the PNG format
Intel Ethernet Drivers	2.17.4	50	64380	Linux kernel drivers for all Intel Ethernet adapters
PCS's Driver	-	20	5483	Driver of a PCS of a railway

Table 5.6 summarizes the results obtained by the tool when testing these applications. The symbol (-) in the table means that the tool could not finish its entire process. Note that some of these applications are early-stage development versions and sometimes present some problems or even are incomplete. In this case, it was necessary to modify some files manually because some include directives were incorrect.

For the *sSocks* application, the tool was able to detect 10 potential vulnerabilities but was unable to proceed with the process because the parser could not parse some files due to problems with includes that were not provided with the program. In the case of *Intel Ethernet Drivers*, the tool detected 1 potential vulnerability but was unable to process the file for the same problem.

In the case of the *LIBPNG* and *Macgen* applications, the tool did not find any potential vulnerabilities associated with the addressed functions. Therefore, it did not generate any fixes.

Table 5.6: Summary of the evaluation of the applications.

Application	Potential Vulnerabilities	Fixes Generated
Zervit	6	4
Macgen	0	0
sSocks	10	(-)
Tiny HTTPd	38	2
LIBPNG	0	0
Intel Ethernet Drivers	1	(-)
PCS's Driver	4	0

For the *Zervit* and *Tiny HTTPd* applications, the tool detected 6 and 38 potential vulnerabilities, respectively. For the 6, it generated 4 fixes, and for the 38, it generated 2 fixes. These fixes were manually verified, and we concluded that they were necessary and correctly removed the vulnerabilities found. This result shows that it was possible to discover 6 vulnerabilities that were not yet reported for these versions of the applications, i.e., the tool discovered 6 zero-day vulnerabilities. The remaining 38 potential vulnerabilities found were in fact not vulnerable, which the Program Validator confirmed, thus invalidating the 38 false positives provided by the Vulnerability Identifier.

Regarding the *PCS's driver* (the partner code), the Vulnerability Identifier detected 4 potential vulnerabilities. However, the Vulnerability Corrector did not generate any fixes because it did not consider these hits real vulnerabilities, as well as the Program Validator. Again we checked these results manually, and indeed all 4 hits were not vulnerable.

Given these results, we can positively answer **Q7**.

Chapter 6

Conclusion

In this work, we studied examples of buffer overflow vulnerabilities in C programs and examined some functions from the C language considered insecure against this class of vulnerabilities. We also analyzed secure versions of these functions and solutions for creating secure code using them, considering aspects for their correct use. In addition, we analyzed several tools and work done related to vulnerability detection and exploitation and automatic software repair. With this analysis, we realized that some of the tools available for detecting vulnerabilities and correcting code generate corrections and apply them to the code without verifying if they are correct and safe. Thus, we proposed a fully automated solution to detect vulnerabilities, confirm their existence, fix the real vulnerabilities, and verify that the fixes generated are effective.

We implemented a prototype of the proposed solution in combination with using two open-source tools, Flawfinder and AFL, which helped the implementation of two modules.

The developed prototype was evaluated using a dataset generated with test cases collected from SARD and real applications collected from SourceForge and a partner of the project this work is inserted.

The experimental results showed that the tool was able to detect vulnerabilities related to buffer overflows and correct them effectively. All the fixes applied by the tool were correctly generated, the code was syntactically correct, and the existing vulnerabilities were successfully removed.

Based on these results, we conclude that our solution satisfies the objectives proposed for this work and could be an asset for work related to automatic software repair. Furthermore, it can be a helpful tool for improving code quality and software security.

6.1 Future Work

In this section, we present some aspects that could be improved in the tool and future directions for continuing the work.

An aspect that could be improved would be to increase the parsing capability of the tool to cover the more complex data structures in the language, which would increase the ability to detect and fix vulnerabilities.

Another aspect would be to improve the tool's output to make it easier for the user to find the problematic places in the files and help perceive existing problems.

A direction for future work would be to add a feature to check if there is any use of the functions that are considered safe (e.g., `strncpy`) and verify if these functions are used correctly to avoid the

problems that we discussed in the analysis of these functions.

Another direction for future work would be to add more sensitive sinks associated with buffer overflows to increase the scope of functions covered by the tool.

Bibliography

[1] American Fuzzy Lop (AFL). https://github.com/google/AFL. [Accessed: 2021-07-10].

[2] CVE-1999-0002 : Buffer overflow in NFS mountd. https://www.cvedetails.com/cve/CVE-1999-0002/. [Accessed: 2022-01-18].

[3] CVE-2004-0234 : Multiple stack-based buffer overflows. https://www.cvedetails.com/cve/CVE-2004-0234/. [Accessed: 2022-01-10].

[4] CVE-2011-1270 : Buffer overflow in Microsoft PowerPoint 2002 SP3. https://www.cvedetails.com/cve/CVE-2011-1270/. [Accessed: 2022-01-18].

[5] CVE-2020-25583: In FreeBSD 12.2-STABLE. https://www.cvedetails.com/cve/CVE-2020-25583/. [Accessed: 2022-01-22].

[6] CWE VIEW: Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/data/definitions/1337.html. [Accessed: 2021-12-10].

[7] Flawfinder. https://github.com/david-a-wheeler/flawfinder. [Accessed: 2021-07-05].

[8] Honggfuzz. https://github.com/google/honggfuzz/. [Accessed: 2022-01-17].

[9] Internet Security Glossary. https://www.ietf.org/rfc/rfc4949.txt. [Accessed: 2022-01-16].

[10] ISO/IEC 27000:2018. https://www.iso.org/standard/73906.html. [Accessed: 2022-01-17].

[11] LibFuzzer. https://llvm.org/docs/LibFuzzer.html. [Accessed: 2022-01-10].

[12] MITRE CWE and CERT Secure Coding Standards. https://www.cisa.gov/uscert/bsi/articles/knowledge/coding-practices/mitre-cwe-and-cert-secure-coding-standards#CERT%20secure%20coding. [Accessed: 2021-07-13].

[13] NIST Software Assurance Reference Dataset Project. https://samate.nist.gov/SARD/. [Accessed: 2022-01-15].

[14] Pycparser. https://github.com/eliben/pycparser. [Accessed: 2021-05-11].

[15] TIOBE Index for December 2021. https://www.tiobe.com/tiobe-index/. [Accessed: 2021-12-05].

[16] Vulnerability - Glossary - CSRC. https://csrc.nist.gov/glossary/term/vulnerability. [Accessed: 2022-01-17].

[17] Chris Anley, Jack Koziol, Felix Linder, and Gerardo Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, Inc., 2007.

[18] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *In Proceedings of the ACM on Programming Languages*, 2019.

[19] El Habib Boudjema, Christèle Faure, Mathieu Sassolas, and Lynda Mokdad. Detection of security vulnerabilities in C language applications. *Security and Privacy*, 2017.

[20] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the International Conference on Software Engineering*, page 122–131. IEEE Press, 2013.

[21] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, page 2095–2108. Association for Computing Machinery, 2018.

[22] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *Proceedings of the 28th USENIX Conference on Security Symposium*, page 1967–1983, 2019.

[23] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Using Sequence-to-Sequence Learning for Repairing C Vulnerabilities. *arXiv*, 2019.

[24] Matt Conover. w00w00 on Heap Overflows. https://www.cgsecurity.org/exploit/heaptut.txt. [Accessed: 2022-02-25].

[25] William Arild Dahl, Laszlo Erdodi, and Fabio Massimo Zennaro. Stack-based Buffer Overflow Detection using Recurrent Neural Networks. *arXiv*, 2020.

[26] Sun Ding, Hee Beng Kuan Tan, and Hongyu Zhang. Automatic removal of buffer overflow vulnerabilities in C/C++ programs. *In Proceedings of the 16th International Conference on Enterprise Information Systems*, 2:49–59, 2014.

[27] Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. Prioritizing Alerts from Multiple Static Analysis Tools, using Classification Models. *International Workshop on Software Qualities and their Dependencies*, 2018.

[28] Fengjuan Gao, Linzhang Wang, and Xuandong Li. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. *In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 786–791, 2016.

[29] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019.

[30] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20–27, 2012.

[31] Gustavo Griecox, Luis Grinblatx, Lucas Uzalx, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using Machine Learning. *In Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, 2016.

[32] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Conference on Security*, page 49–64. USENIX Association, 2013.

[33] Eric Haugh and Matt Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. *In Proceedings of the Symposium on Network and Distributed Systems Security*, 2003.

[34] João Inácio and Ibéria Medeiros. Effectiveness on C Flaws Checking and Removal. *In Proceedings of the 52nd IEEE/IFIP International Conference on Dependable Systems and Networks*, 2022.

[35] Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*, 2021.

[36] Jorrit Kronjee. Discovering Software Vulnerabilities Using Data-Flow Analysis and Machine Learning. Master's thesis, Open University, faculty of Management, Science and Technology, 2018.

[37] Zhen Li, Deqing Zou, Shouhuai Xux, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *In Proceedings of the Network and Distributed Systems Security Symposium.*, 2018.

[38] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, page 329–340, 2007.

[39] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 101–114, 2020.

[40] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering*, 2016.

[41] Martin Monperrus. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.*, 51(1), 2018.

[42] Ricardo Morgado, Ibéria Medeiros, and Nuno Neves. Towards web application security by automated code correction. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2020.

[43] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy*, 3(2):58–62, 2005.

[44] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 1996.

[45] Krerk Piromsopa and Richard J. Enbody. Survey of Protections from Buffer-Overflow Attacks. *Engineering Journal*, 15:31–52, 2011.

[46] Rebecca L. Russell, Louis Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *arXiv*, 2018.

[47] Arthur D. Sawadogo, Tegawendé F. Bissyandé, Naouel Moha, Kevin Allix, Jacques Klein, Li Li, and Yves Le Traon. Learning to Catch Security Patches. *arXiv*, 2020.

[48] Robert C. Seacord. *Secure Coding in C and C++*. The SEI Series in Software Engineering. Addison-Wesley, 2013.

[49] Hossain Shahriar, Hisham M. Haddad, and Ishan Vaidya. Buffer Overflow Patching for C and C++ Programs: Rule-Based Approach. *ACM SIGAPP Applied Computing Review*, 13(2):8–19, 2013.

[50] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic Error Elimination by Horizontal Code Transfer across Multiple Applications. *ACM SIGPLAN Notices*, 50(6):43–54, 2015.

[51] Alexey Smirnov and Tzi-cker Chiueh. Automatic Patch Generation for Buffer Overflow Attacks. In *Proceedings of the 3rd International Symposium on Information Assurance and Security*, pages 165–170, 2007.

[52] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., 2008.

[53] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural Program Repair by Jointly Learning to Localize and Repair. *In Proceedings of the International Conference on Learning Representations*, 2019.

[54] Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles. Detection of Security Vulnerabilities in C Code using Runtime Verification: an Experience Report. *In Proceedings of the International Conference on Tests And Proofs*, 2018.

[55] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling Black-Box Mutational Fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, page 511–522, 2013.

[56] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning. *In Proceedings of the USENIX Workshop on Offensive Technologies*, 2011.

[57] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical report, Departement Computerwetenschappen, Katholieke Universiteit Leuven, 2004.

[58] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[59] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2019.