UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Automatic Binary Patching for Flaws Repairing using Static Rewriting and Reverse Dataflow Analysis

Diogo Tomás Ferreira

Mestrado em Segurança Informática

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

2022

Acknowledgments

Firstly, I would like to thank my advisor, Prof. Ibéria Medeiros for all the support, guidance and time invested in me and this dissertation.
To my wife Sónia for all the support she has given me throughout the years. For helping me stay motivated and focused on my goal while taking care of our newborn baby girl, Matilde.
To my mother, Conceição, father, Álvaro, and sister, Joana, for all the unconditional support over the years, I could not have made it without you all.
Finally, a special appreciation to my grandmother, Isabel, who raised me to be who I am today and to whom I had promised that I would finish this dissertation.
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).

I hope you are up there, proud of your grandson. Thank you.

Resumo

O desenvolvimento de *software* cresceu significativamente nos últimos anos, tendo este sido também acompanhado pelo aumento do código fonte produzido, não só em tamanho mas também em complexidade. O aparecimento de novas evoluções digitais, tal como a indústria 4.0 e o aumento do número de dispositivos utilizados no nosso dia a dia, contribuíram para a constante necessidade de novas funcionalidades e desenvolvimentos mais rápidos e capazes de acompanhar a procura, aumentando assim a complexidade do *software* produzido e favorecendo o aparecimento de vulnerabilidades e falhas nos produtos finais das empresas. Ao longo dos anos foram surgindo diversas vulnerabilidades que, quando exploradas, causaram impactos negativos significantes. Por outro lado, a sua exploração despoletou o aumento da sensibilização para a necessidade de olhar seriamente para segurança implementada no código do *software* que corre nestes dispositivos, tornando-se importante não só detetar, mas também mitigar atempadamente as vulnerabilidades.

A linguagem de programação C é muito utilizada na construção de sistemas computacionais e sistemas embebidos, seja na programação do *kernel* ou dos programas que nestes correm, fazendo desta linguagem uma das linguagens mais utilizadas e, por isso, também, a que tem mais vulnerabilidades reportadas nos últimos dez anos. Devido à sua natureza de baixo nível e reduzida abstração para o programador, as vulnerabilidades de corrupção de memória são um dos tipos mais comuns em C, especificamente as relacionadas com *buffers* que ocorrem quando uma aplicação falha em assegurar que uma operação de escrita ou leitura não excede a fronteira limite de um *buffer*. Este tipo de vulnerabilidades, quando exploradas, podem ter consequências severas, sendo por isso de extrema importância a sua deteção e mitigação.

Existem diversas ferramentas capazes de detetar vulnerabilidades e possíveis *buffer overflows* que recorrem ao uso de análise estática ou análise dinâmica. A análise estática de código, apesar de conseguir detetar pontos de entrada destas vulnerabilidades, tem uma baixa eficácia e pode produzir um elevado volume de falsos positivos. A análise dinâmica consegue identificar os pontos resultantes da exploração (i.e., pontos sensíveis) destas vulnerabilidades, mas falha na obtenção da sua origem (i.e., pontos de entrada). Também, a mitigação automática de vulnerabilidades é um tópico que tem recebido bastante atenção, no entanto, tal como acontece com as ferramentas de deteção de vulnerabilidades, a maio-

ria destas requer o acesso ao código fonte da aplicação e/ou que esta seja compilada com o seu código instrumentalizado de modo que as ferramentas possam ser executadas com sucesso. As poucas ferramentas de reparação de código que conseguem lidar somente com o código binário (código fonte compilado, originando assim o código do objeto) dos programas, compilados sem qualquer instrumentalização de código, requerem normalmente elevados recursos do sistema, sendo a sua utilização incompatível com um ambiente de sistema embebido, onde os recursos são limitados.
O principal foco desta dissertação é o desenvolvimento de uma ferramenta capaz de automaticamente detetar e corrigir vulnerabilidades de *stack overflow*, assim como verificar a eficácia da correção, sem que para isso seja necessário aceder ao código fonte da aplicação ou qualquer forma prévia de instrumentalização do código binário da aplicação em análise. Recorrendo a técnicas de *fuzzing*, ao código binário não instrumentalizado, e a um conjunto de casos de teste, a ferramenta gera novos *inputs* na tentativa de expor novos caminhos de execução no código binário para encontrar/explorar vulnerabilidades, guardando para cada uma, o *exploit* que permitiu a sua exploração. Após esta primeira fase, a ferramenta para cada vulnerabilidade encontrada e explorada com sucesso, utiliza o respetivo *exploit* e executa uma análise dinâmica na execução do código binário num ambiente de *debug* que possibilita a regressão na execução da aplicação, desde o momento em que aplicação quebra até à origem, i.e., o ponto sensível (*sensitive sink*). Numa terceira fase, e após identificada a origem da vulnerabilidade, esta ferramenta irá proceder à correção da mesma recorrendo à técnica de reescrita estática de código (*static re-writer*) com trampolins, que permite a adição segura de código em zonas de memória livres, assegurando a integridade do código binário sem a necessidade de recuperar o *control flow* original. A correção é feita recorrendo a *templates* escritos em C e previamente compilados, produzidos especificamente para a mitigação de funções vulneráveis, substituindo-as pelas respetivas versões seguras ou limitando os parâmetros de entrada destas funções de modo a assegurar que os limites dos *buffers* são respeitados, e produzindo um novo ficheiro binário isento das vulnerabilidades encontradas. Numa quarta fase, a ferramenta testa o novo ficheiro binário de modo a garantir que o código inserido não adicionou novas vulnerabilidades. Para tal, a ferramenta recorre à técnica de *fuzzing*, aos *exploits* que despoletaram as vulnerabilidades agora corrigidas, e a novos casos de teste gerados a partir desses *exploits*. Caso não sejam encontradas quaisquer vulnerabilidades no final desta sessão, o ficheiro binário é considerado corrigido com sucesso.
A ferramenta foi testada com um conjunto de pequenos programas criados para o efeito, bem como um conjunto de aplicações reais. Os resultados das experiências mostraram que a ferramenta é eficaz na deteção e correções de vulnerabilidades de *stack overflow*.

Palavras-chave: Vulnerabilidades de Stack Overflow, Correção de Código Binário, Engenharia Reversa, Análise Dinâmica, Segurança de Software

Abstract

The C programming language is widely used in embedded systems, kernel and hardware programming, making it one of the most commonly used programming languages. However, C lacks of boundary verification of variables, making it one of the most vulnerable languages. Because of this and associated with its high usability, it is also the language with most reported vulnerabilities in the past ten years, being the memory corruption the most common type of vulnerabilities, specifically buffer overflows. These vulnerabilities when exploited can produce critical consequences, being thus extremely important not only to correctly identify these vulnerabilities but also to properly fix them. This work aims to study buffer overflow vulnerabilities in C binary programs by identifying possible malicious inputs that can trigger such vulnerabilities and finding their root cause in order to mitigate the vulnerabilities by rewriting the binary assembly code and thus generating a new binary without the original flaw.

The main focus of this thesis is the use of binary patching to automatically fix stack overflow vulnerabilities and validate its effectiveness while ensuring that these do not add new vulnerabilities. Working with the binary code of applications and without accessing their source code is a challenge because any required change to its binary code (i.e, assembly) needs to take into consideration that new instructions must be allocated, and this typically means that existing instructions will need to be moved to create room for new ones and recover the control flow information, otherwise the application would be compromised.

The approach we propose to address this problem was successfully implemented in a tool and evaluated with a set of test cases and real applications. The evaluation results showed that the tool was effective in finding vulnerabilities, as well as in patching them.

Keywords: Stack Overflow Vulnerabilities, Binary Patching, Reverse Engineering, Dynamic Analysis, Software Security

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development has grown significantly in recent years and, as often occurs with fast pacing industries, in most cases it has looked at security as an afterthought. The emergence of new digital evolution, such as Industry 4.0 [8], and the increase of the number of devices in our everyday life, has pushed for a constant need of new features and faster developments, and as a result, software has become more complex which favors the appearance of bugs and vulnerabilities. Over the years, the appearance of such vulnerabilities [17][19] has raised awareness of the need to take security seriously when it comes to the software running on these devices.

Vulnerabilities and bugs in the source code of applications occur during the development phase and, in the case of embedded systems, they are mostly inserted due to the usage of low level programming languages that are more prone to emergence of unintentional errors introduced by programmers and the lack of safe programming principles. Companies have limited budgets and resources for any given software project and usually neglect software security for the implementation of new features and functionalities, and security assessments occur after the software is already in production, leaving it exposed and vulnerable to attacks.

Given the high demand for new applications and their faster availability in the market, there is a need for tools that are capable of finding vulnerabilities in software, either by analyzing its source code or the produced binary. However, finding a vulnerability is only half of the work, since it also have to be mitigated, i.e., moved, requiring either more code development or another set of tools that are able to automatically patch the code to remove the vulnerability. Although there are already several tools that can help with the detection task, there are fewer tools for vulnerability repairing. For these last, most of them are focused on source code repairing, and very few are able to handle binary patching. Those that are able, they have several limitations such as high system requirements and resources, incompatibility with the limited environment that usually exists for embedded systems, and the necessity of code instrumentation to proceed with binary code analysis, but, for this, it is necessary to access the source of the program, something that is not always available or allowed for software testing.

1.1 Motivation

The demand for software development over the last two decades have been steadily growing, increasing the source code not only in size and complexity, but also in the number of weaknesses and flaws that will most likely be present in the final product. These weaknesses are commonly known as vulnerabilities and if not detected and fixed can usually impact performance and more importantly the security of the software.
C language is one of the most commonly used programming language in the development of software, specially key elements of embedded systems, kernel and hardware programming are built using C as the primary language, which is one of the reasons why C is the language with most reported vulnerabilities in the past 10 years [10]. Due to its low level nature and the ability of allowing the programmer to be responsible for managing and allocating memory, memory corruption is the most common type of vulnerabilities in C [22], specifically buffer errors and overflows. These vulnerabilities when exploited can cause critical consequences, which makes the identification and mitigation of these type of vulnerabilities extremely important.
There are several tools that can detect the possible existing buffer overflows in programs, either by static analysis [16], which can detect possible entry points for vulnerabilities but has low precision since it can generate a high number of false positives, or by dynamic analysis [9], which can identify the point where a crash occurred, but it fails to pinpoint the root cause of the problem - the sensitive sink point. The automatic repairing of a vulnerability is also a topic that has received attention, but most of the existing tools require access to the source code or binary instrumentation in order to the repair task takes place, which is not always available specially for production closed source binaries.

1.2 Objectives

The focus of this dissertation is the development of a tool that is capable of detecting and repairing buffer overflow vulnerabilities in C programs and verify the effectiveness of the applied corrections, without accessing their source code and without any form of code instrumentation prior to its compilation. To pursuit with such tool and main goal, we can divide this goal into the following five:

- The first objective is to study some of the most common stack overflow vulnerabilities in C language, specifically the different function that are considered insecure against these type of vulnerabilities. Also, their safe versions (if exist) will be studied.

• The second objective is to study different techniques and determine the best one to search for vulnerabilities in a non-instrumentalized binary of a program.

• The third objective is to dynamically explore an application during its execution, determining the best techniques to obtain the vulnerability sensitive sink.

• The fourth objective is to study different techniques for performing binary patching that do not require neither prior binary instrumentation nor access to the source code, and that are able to effectively fix a vulnerability.

• The fifth objective is to design and implement a tool based on the information obtained in the previously mentioned studies, evaluating its performance and effectiveness.

1.3 Contributions

The main contributions of this dissertation are the following:

• A study of stack overflow vulnerabilities in C language and the different insecure functions that are associated with these type of vulnerabilities.

• An approach to search for vulnerabilities, obtaining its exploits and determining their sensitive sink by analyzing a non-instrumentalized binary application during its execution and without accessing the source code. Perform a binary patch, able to effectively fix a vulnerability on a non-instrumentalized binary and without accessing the application source code. This approach also needs to be able to validate the effectiveness of an applied patch, confirming that a vulnerability is fixed.

• A code repair approach that is able to modify the assembly code of a binary without changing the application internal logic, resorting for that the trampoline concept and the static re-write technique.

• A tool that implements the proposed approach.

• An experimental evaluation of the tool, using synthetic test cases and real applications, to assess its performance and its effectiveness.

1.4 Document structure

This document is organized as follows:

• Chapter 2 defines key concepts that are relevant for this project, such as software vulnerabilities, buffer overflows and fuzzing. It also presents several existing tools like AFL and GDB that will be used in this project.

- Chapter 3 presents the proposed solution and details the components that integrate it.
- Chapter 4 details the implementation of the proposed solution.
- Chapter 5 evaluates the proposed solution.

Chapter 2

Context and Related Work

This chapter presents the necessary context to support the goal of this dissertation and discusses relevant related work, more specifically it gives an overview of the vulnerabilities we focused (Section 2.1), the current techniques for their detection (Section 2.2), how to track and instrumentalize the binary code (Sections 2.3 and 2.4) and the security mechanisms that can be employed in binary code (Section 2.5). This chapter ends with the discussion of the related work for detection and repairing over binary code (Section 2.6)

2.1 Software vulnerabilities

During the development of software, one of the phases of its development cycle is testing, which aims to find weaknesses or flaws in the source code. According to Microsoft Security Response Center [43], these weaknesses, also known as vulnerabilities, could allow an attacker to compromise the integrity, availability or confidentiality of the software.
Vulnerabilities can be a direct result of a wide variety of factors, such as importing a vulnerable third-party library or poorly implemented application logic, which makes the task of finding these very difficult, even with manual code reviews and proper testing solutions.
Although there is a wide array of vulnerabilities, the range of categories in which they can fall into is quite limited but extremely important to allow for quick understanding of its impact in the software when exploited. The categories can range from memory corruption to input validation errors or race conditions, among others.

2.1.1 Buffer overflows in C

One of the most common and oldest software vulnerability is the buffer overflow, which happens when an application fails to bound check a writing/reading operation to a buffer, allowing it to overwrite/read outside its boundaries. It is considered the most common and impactful vulnerability.

Buffer overflows can happen either in the heap [31], a memory region used to store dynamic values, or in the stack [31], a special memory region used to store local variables as well as function parameters and their return addresses. Depending on the region where the buffer overflow occurs, it can be referred to as Heap Overflow or Stack Overflow[29], respectively. Younan et al. did a survey on implementation errors in C and C++ [52], focused on those errors that allow an attacker to break memory safety and execute code. The authors analyzed five types of vulnerabilities - dangling pointer references, heap-based and stack-based buffer overflows, format string and integer errors, as well as existing proposed countermeasures that try to mitigate these vulnerabilities, examining its effectiveness against the exploitation of these common implementation errors. In order to better study the effectiveness of such mitigations, all countermeasures were divided into several categories based on the vulnerabilities they address and the type of protection they offer, their limitations in terms of applicability and protection and the type of response once the problem is detected. This division allowed the authors to conclude that most of the analyzed countermeasures take an ad-hoc approach when it comes to mitigate or prevent the impact of specific vulnerabilities and the need of designing a model that would describe key abstractions that the compiler can rely on to generate the program.

```c

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

void secret() {
    printf("Stack overflow detected!\n");
    exit(0);
}

void test(char *s) {
    char buf[10];
    strcpy(buf, s);
}

int main (int argc, char** argv) {
    test(argv[1]);
    printf("No stack overflow detected!\n");
    return 0;
}
```

Listing 2.1: Stack overflow example

The C code above (Listing 2.1) demonstrates a simple stack overflow, where the test function creates a 10 byte buf array and performs a strcpy operation on that array without any kind of boundary check. This means an attacker can exploit this code to overwrite a memory register in order to execute other functions that are not in the scope. In this example, the goal is to execute the secret function by overwriting the return address of the test function.

```
1   test:
2
3       endbr64
4       pushq       %rbp
5       movq        %rsp, %rbp
6       subq        $32, %rsp
7       movq        %rdi, -24(%rbp)
8       movq        -24(%rbp), %rdx
9       leaq        -10(%rbp), %rax
10      movq        %rdx, %rsi
11      movq        %rax, %rdi
12      call        strcpy@PLT
13      nop
14      leave
15      ret
16      .size       test, .-test
17      .section    .rodata
```

Listing 2.2: Test function assembly code

Converting the C code to assembly, a low level programming language that is converted to machine machine code by the assembler, we can see how the test function is built and its memory representation, as seen in code in Listing 2.2.

After the RBP value is saved in the stack and assigned to it the RSP registers (lines 4 and 5), 32 bytes are allocated for this function (line 6) and the buf variable is given a 10 bytes allocation right bellow the base pointer register - RBP (line 9), which points to the base of the current stack frame. Each register has a size of 8 bytes and above the RBP is the rip, the instruction pointer, containing the address of the next instruction to be processed. A stack overview can be seen in Figure 2.1.

This can be verified using a debugger, in this case GDB [35], to get the register value at the time the application crashes given the input "AAAAAAAAAABBBBBBBBCCCCCC", which will fill the buf with A's, then the register RBP with B's and finally the register rip with C's (Figure 2.2). Note that although x86-64 registers are 64 bits, because of the range of the virtual address space only 48 bits are actually used, the low half of the 48 bit, leaving the top 2 bytes of a return address set to zero, and this is why the rip register is only overwritten with 6 C's, which is 6 bytes (48 bits).

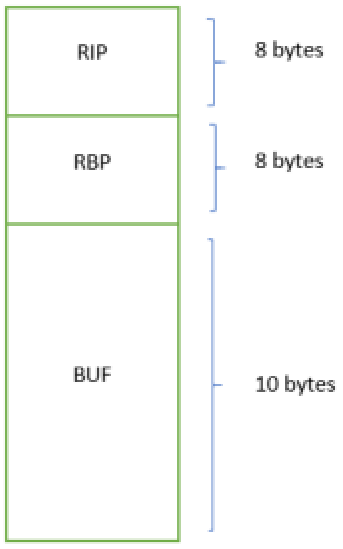
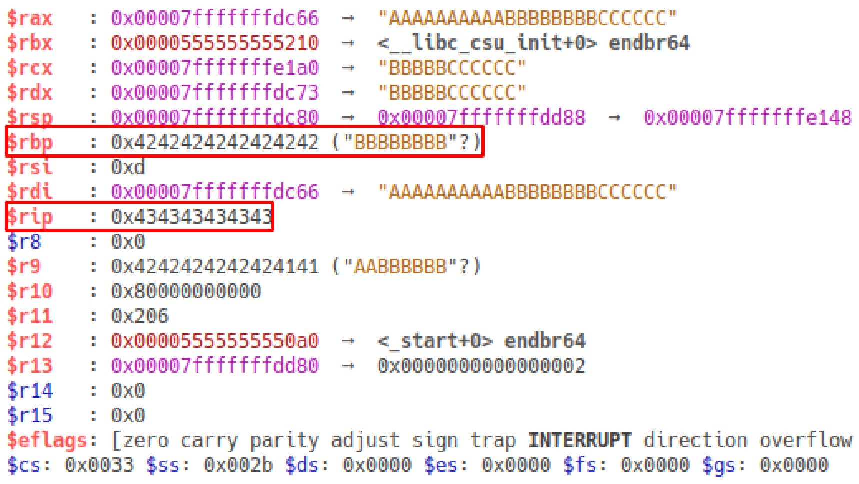


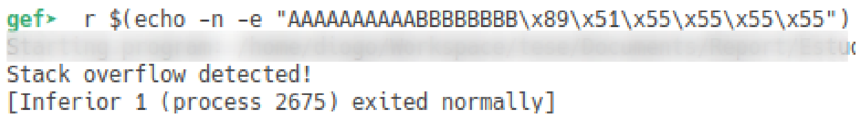
Figure 2.1: Stack overview.



```
$rax   : 0x00007fffffffdc66  →  "AAAAAAAAAABBBBBBBBCCCCCC"
$rbx   : 0x0000555555555210  →  <__libc_csu_init+0> endbr64
$rcx   : 0x00007fffffffe1a0  →  "BBBBBCCCCCC"
$rdx   : 0x00007fffffffdc73  →  "BBBBBCCCCCC"
$rsp   : 0x00007fffffffdc80  →  0x00007fffffffdd88  →  0x00007fffffffe148
$rbp   : 0x4242424242424242 ("BBBBBBBB"?)
$rsi   : 0xd
$rdi   : 0x00007fffffffdc66  →  "AAAAAAAAAABBBBBBBBCCCCCC"
$rip   : 0x434343434343
$r8    : 0x0
$r9    : 0x4242424242424141 ("AABBBBBB"?)
$r10   : 0x80000000000
$r11   : 0x206
$r12   : 0x00005555555550a0  →  <_start+0> endbr64
$r13   : 0x00007fffffffdd80  →  0x0000000000000002
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
```

Figure 2.2: Register values when crashing

This example of code is vulnerable to stack overflow and an exploitation path can be to overwrite the the rip register with the secret function address in order to get this function to run. For that to happen an attacker would have to know the address of the function secret, which in this case we will assume it is 00x555555555189. Using this address and modifying the initial input argument to "AAAAAAAAAABBBBBBBB\x89\x51\x55\x55\x55\x55" as shown in Figure 2.3 - Note the fact that the address is in reverse order because this was performed on a little-endian machine [44].



```
gef➤  r $(echo -n -e "AAAAAAAAAABBBBBBBB\x89\x51\x55\x55\x55\x55")
[redacted]
Stack overflow detected!
[Inferior 1 (process 2675) exited normally]
```

Figure 2.3: Stack overflow result

2.2 Vulnerability detection

There are several security techniques that can be implemented in order to detect software vulnerabilities, and two of the most widely used are static and dynamic analysis.c Static analysis [46] is commonly used in software development because it allows for a fast code auditing with a certain degree of abstraction, making it very useful in software development, specifically when it involves millions of code lines which makes manual code review impractical. This type of analysis is performed in a non-runtime environment, testing and evaluating an application source code by looking for code patterns that may indicate a vulnerability. This is why this type of analysis is heavily dependent on an updated set of patterns and rules in order for it to work, otherwise the resulting accuracy and precision will suffer with a high level of false positives and negatives. Dynamic analysis [46] on the other hand is performed at runtime, evaluating the application running state and manipulating it in order to discover new vulnerabilities.Dynamic analysis, although capable of exposing complex vulnerabilities that would not be found with static analysis, it will only be able to analyze parts of the code that are actually executed. In order to ensure that most of the code is executed, it is important to have the maximum code coverage possible, and so having the right input tests cases is a necessity.
Vadayath et al. [51] studied current state-of-the-art binary program analysis approaches and have identified a set of vulnerability properties that allow to improve the precision of static vulnerability detection and the scalability of dynamic vulnerability detection. The authors created a prototype called ARBITRER, an hybrid analysis technique that is able to analyze large amounts of binary code, with high precision, even for complex vulnerabilities such as intricate occurrences of integer overflows or privilege escalation bugs. ARBITRER was evaluated on 76,516 binary programs and its effectivess was successfully demonstrated on four common vulnerability classes, CWE-131 (Incorrect Calculation of Buffer Size), CWE-252 (Unchecked Return Value), CWE-134 (Uncontrolled Format String) and CWE-337 (Predictable Seed in Pseudo-Random Number Generator).

2.2.1 Fuzzing

One the most widely used techniques in quality assurance and security testing is fuzzing, which tries to achieve maximum code coverage by generating new test cases that aim to find and explore new code paths, constantly monitoring the application in order to find errors and possible crashes. In general, all fuzzers commonly implement at least three main components:

1. Generator - Responsible for generating test cases/inputs that will be used to test the system. This generators can either be grammar based or mutational based.

2. Delivery Mechanism - Responsible to deliver the generated inputs/test cases to the

system.

3. Monitoring System - Responsible for monitoring the system while it runs in order to detect errors or crashes, that might indicate a vulnerability.

Although there are many different fuzzers, we can categorized them all into three main types:

- **Blackbox** fuzzers have no prior knowledge about the testing environment and are able to generate random inputs that are then used against the system. Although this type of fuzzers are extremely quick to run and required almost no setup due to its random generated inputs they can take a lot of time to find new bugs.

- **Whitebox** fuzzers are the complete opposite of Blackbox fuzzers, requiring well-formed test cases that will be used against the system leveraging program analysis to find and explore new paths. This allows white fuzzers to be very effective at finding new bugs and issues deep within the code but at the cost of the time needed in the analysis phase, which can sometimes be prohibitive.

- **Greybox** fuzzers are in between both Whitebox and Blackbox fuzzers. Rather than have to perform program analysis, these type of fuzzers require lightweight instrumentation in order to glean information about a system internal structure, allowing it to be more effective than blackbox fuzzing and more efficient than whitebox fuzzing.

Below is some related work that uses fuzzing to find vulnerabilities. Fioraldi et al. [33] have identified a big gap between fuzzing frameworks and cutting edge techniques because of the lack of an API that would allow for a smooth integration. This problem impacts not only the industry since there is no easy way of setting up a new research and so it is hard to evaluate which one is worth the attention, but it also impacts the researchers themselves because of how hard it is to evaluate their tools and to combine functionality with with compatible techniques. This is why AFL++ was developed, to provide a fuzzing framework that gives researchers access to an extensive API to build upon and so allowing them to evaluate combinations of their proposals with a highly reduced implementation effort, but also to allow the industry access to a set of easy-to-use features that came straight from cutting-edge research which can greatly benefit the outcome of a fuzzing campaign.
Lyu et al.[42] present a novel mutation scheduling scheme MOPT, which enables mutation based fuzzers to discover vulnerabilities more efficiently. Mutation-based fuzzing is one of the most popular vulnerability discovery solutions, and although its performance of generating interesting test cases highly depends on the mutation strategy, fuzzers usually

follow a specific distribution to select mutation operators, which is inefficient in finding vulnerabilities on general programs. MOPT utilizes a customized Particle Swarm Optimization (PSO) algorithm to find the optimal selection probability distribution of operators with respect to fuzzing effectiveness, and provides a pacemaker fuzzing mode to accelerate the convergence speed of PSO. Being one of the most cutting-edge mutation scheduling schemes, MOPT is also the one used by default in AFL++.

Although genetic algorithm-based fuzzing is able to mutate the seed files provided by the users to create several new inputs that will later be used to test the target application in order to try to trigger potential crashes, the current seed selection strategies do not seem to be better than randomly picking seed files. In this paper [41], Lyu et al. propose a generic system, named SmartSeed, to generate seed files towards efficient fuzzing, leveraging machine learning model to learn and generate high value binary seeds. SmartSeed was evaluated along with AFL against 12 open-source applications and was able discovered more than twice unique crashes when compared to other existing seed selection strategies, which resulted in a total of 16 new vulnerabilities that have received CVE IDs [1].

Ispoglou et al. presented FuzzGen [38], a new tool for fuzzing libraries. Because libraries cannot run as standalone programs, and instead are invoked through another application, which means that triggering code deep in a library remains challenging, since a specific sequence of API calls are required to build up the necessary state. FuzzGen goal is to automatically synthesizing fuzzers for complex libraries in a given environment, leveraging a whole system analysis to infer the library's interface and thus synthesizing specific fuzzers for the given library without human interaction. This generated fuzzers are able not only to achieve better code coverage but also to expose bugs that reside deep in the library.

Rustamov et al. have designed a hybrid fuzzing tool, DeepDiver [49], that aims to discover vulnerabilities deep within the code while negating roadblock checks, a limitation found in other existing hybrid fuzzing frameworks, and thus allowing the fuzzer to explore new execution paths that can trigger vulnerabilities deep withing the binary. DeepDiver combines AFL++[32], which is the newer version of AFL, with concolic execution engine to perform its job. It detects a roadblock check by leveraging a trace analysis approach and building a tree for each input, negating it in the next stage using one of the various methods available - Stelix, which leverages Dynamic Binary Instrumentation or Static Binary Translation which is used to change the control flow graph structure. DeepDiver was tested with the LAVA-M dataset and eight large real-world programs, and showed promising result, outperforming existing software testing tools.

2.2.2 American Fuzzy Lop (AFL)

American Fuzzy Lop - AFL [53], is one the best well known security greybox fuzzers, and it uses a compile-time instrumentation along with a genetic algorithm to discover new tests cases that could trigger new internal states, and thus improving the code functional coverage. AFL was design to be practical, using a highly effective variety of effort minimization tricks and fuzzing techniques, with a decent performance overhead. Radamsa[13].

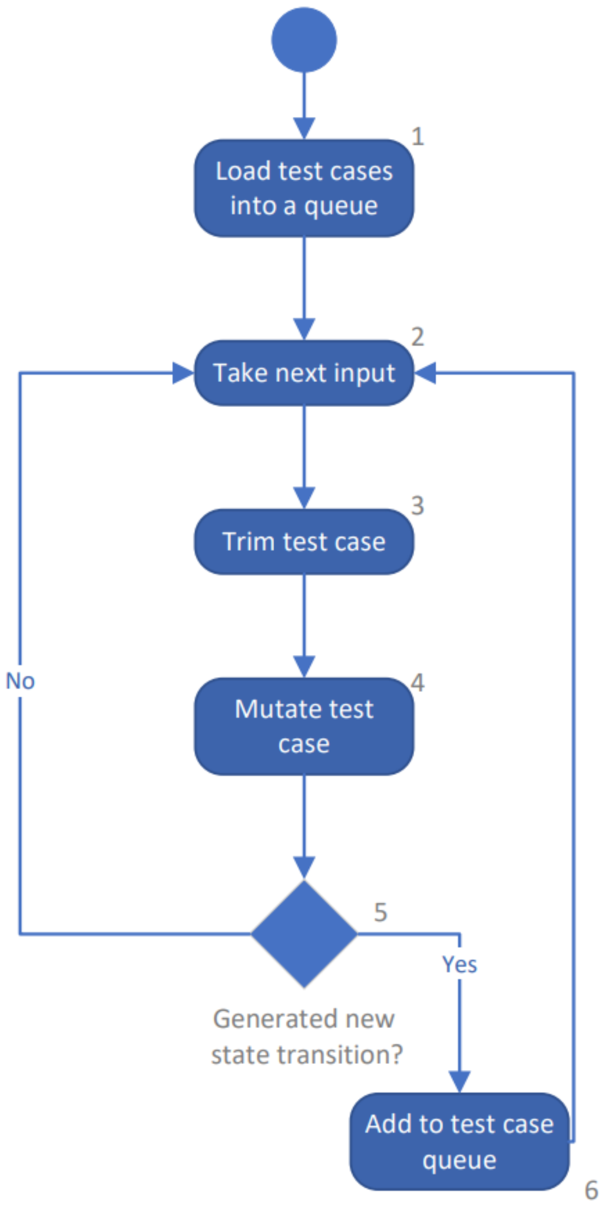


Figure 2.4: AFL control flow

AFL work as described in Figure [2.4] where it starts by loading the initial test cases into a queue (1). Once this first step is completed, it takes one of the input files from the queue (2) and attempts to trim it to the smallest size possible, without altering the measured behaviour of the program (3). It then repeatedly mutates the input file using a variety of fuzzing strategies and mutators (4) and if any of the mutations results in a new program state (5), it is added to the queue (6). A new input is once again taken from the queue and the cycle repeats. Throughout the years, new features and improvements were developed but since no updates were made to the AFL, a new fork emerged - AFL++ [32], that is being maintained and updated with the latest, tested, techniques that improve on the original version. AFL++ supports faster instrumentation modules such as LLVM [36] and QEMU [25], as well as new and better mutators like MOpt[42] and

2.3 GNU Project Debugger (GDB)

Monitoring an application in runtime allows for an opportunity to halt and analyze its execution state in specific points, such as memory contents and CPU registers values. To accomplish this task a debugger tool is needed in order to run the application in a controlled environment and under controlled conditions. The GNU Debugger (GDB) [35] is one of the most well established debugging tools, first written in 1986 by Richard Stallman and maintained nowadays by the Free Software Foundation, it is supported in most Unix-like systems and supports many programming languages such as C, C++, Go, etc. GDB uses ptrace system call (abbreviation of ”process trace”) commonly found in Unix and Unix-like systems, and that allows a process to control another, enabling it to manipulate and inspect the target internal state. This allows GDB to offers a wide variety of facilities for altering and tracing the execution of applications, allowing one not only to monitor the application behavior but also to modify its internal variables values and thus changing its internal state of execution. GDB main features are:

- **Python support:** GDB supports Python scripting, which allows for the automation of GDB debugging tasks.
- **Reversible debugging:** [6] This feature provides the ability to perform ”backward steps” in a debug session. It requires the session the be record [5] before it can be used, so that GDB can track all changes from each executed machine instruction.
- **Watch points:** This gives the ability to set GDB to stop in watch points, when a value is changed. This value can be an address, a variable or a register and allows users to analyze the execution of a program.

2.4 Binary instrumentation

As seen in the Section 2.2.2, in order to work as intended, fuzzers like AFL need previously instrumentalized binaries which almost always requires access to the original source code, so the application can be properly compiled. But what if the source code

is not available? For these cases, there are binary rewriting techniques that can modify the original binary in order to add instrumentation support, and these techniques can be either dynamic [23] and static [30].
Common dynamic rewriting techniques, like the usage of QEMU, usually involves sitting in between the system and the application and acting as a proxy and monitoring all interaction. These techniques are heavier on the system, requiring much more resources to be able to run either the application and the emulator, but are also simpler to implement since they do not involve actually binary modification.
Static binary rewriting techniques, on the other hand, actually create a new binary with the modifications in place. For this, it needs to disassemble the original binary followed by the recovery of the control flow information, the actual transformation that modifies the binary e finally the creation of the new modified binary. The control flow information, however, is a hard problem to solve, specifically for larger applications and one that have been studied with several new techniques emerging to tackle this issue. One of those techniques is instruction punning [27] where a jump instruction is inserted in the binary with the relative jumping address, serving either data and as an instruction sequence and allowing not only for the new code to be added in other places of memory acting as a trampoline, but also requiring no control flow information since the code follows is not affected because after the trampoline code executes it resumes its execution. Instruction punning opened the door for more developments on the binary rewriting and tools like e9patch [30] emerged, where instruction padding and eviction were also used to improve the rewriting performance and reliability on large binaries, as well as patch programming language that allows for an easy patch design and binary instrumentation.

2.5 Security mechanisms

When compiling a C program, there are several security mechanisms that can be used or activated that will hardened overall security of the compiled binary, protecting it against not only buffer overflows but also against code execution. Techniques such as Address Space Layout Randomization (ASLR) [34] that randomizes the location where system executables are loaded into memory, and Data Execution Prevention (DEP) [2] which prevents certain pages or regions of memory from executing code, are offered by the operating system itself in order to prevent memory corruption attacks from executing code, and can be turned on or off independent on the binary.
Other type of security mechanism are the binary techniques, which need to be activated when the application is compiled and can help protect the binary against code execution like Position Independent Executable (PIE) which loads the binary and its dependencies into random locations in the virtual memory every time the applications is executed. Stack canaries [28] are another binary technique that can be used to detect stack smashing, and

it works by generating a value that is placed on the stack between a buffer and control data, i.e, RBP register, which is then monitored and validated in order to detect changes to its original value since it will be the first data to be corrupted when a stack overflow occurs, thus triggering a stack smashing alert and interrupting the execution.
Richarte et al. [47] studied stackshield and stackguard protections, stack shielding technologies that protect programs against stack overflow exploitation by altering the way programs are compiled. These stack protections only protect against return address overwrites and not generic stack smashing, and the authors have identified two design limitations, the fact these only protect data located in higher memory and that these only check for attacks right before a function returns. The authors also identify a technical flaw, which is the fact that they leave the saved frame pointer unprotected. In this paper, the authors described four different tricks to bypass stackshield and stackguard protections, one being a direct consequence of the design limitation and the other three a result of the technical problem.

2.6 Automatic Software Repair

Automatic software repair is considered a hot topic and consists of automatically fixing a software flaw without human intervention. As software becomes more complex and more vulnerabilities are being disclosed, this subject is of great importance since it can help solve these issues. It is however a challenging task since it first needs to correctly identify the right repair and then apply the fix in a way that it does not create new flaws as a result. In the last few years, several techniques have been studied and proposed to address this issue and efficiently repair and maintain software. Below are some works related to automatic software repair.
Shahriar et al. [20] proposed a rule-based approach to identify and mitigate buffer overflows in C/C++ programs, addressing both simple and complex forms of code and ranging from unsafe library function calls to pointer usage in control flow structures. A total of 12 patching rules were proposed by the authors to replace vulnerable code and thus removing vulnerabilities with results showing that these rules were not only able to identify previously known buffer overflow vulnerabilities but also new ones, as well as patch those with a negligible overhead to the application. Schulte et al. [50] proposed a technique for repairing binary programs directly, using evolutionary computation algorithms. The authors focused their efforts on embedded and mobile systems because of its resource constraints and tight coupling with its execution environment, which makes existing techniques and tools insufficient. They were able to demonstrate that assembly and binary repairing was capable of producing the same level of results as source-statement-level repairs, but in a more efficient way, showing that not only it was the repairing faster, but it also needed much smaller disk and memory requirements.

Klieber et al. [40] presented a technique to repair a C program, at the source code level, against potential violations of spatial memory safety, differing from other traditional program repair techniques by focusing on preventing a security vulnerability from being exploited by an attacker. Although many techniques already exist that can harden software against memory related vulnerabilities, many of those create dependencies on the compilers, making it difficult to inspect to fine-tune the repair itself.
The analysis and transformation needed to repair a vulnerability at the source code level is most easily done at an intermediate representation, but existing approaches have fundamental limitations when it comes to translating changes back to source code. This is why the technique presented by the authors tackles this challenge by first translating the intermediate representation to an abstract syntax tree level, using a carefully designed set of transformation rules and repair transformations, where changes can then be translated back to source code level with the help of a modified clang. This approach was implemented in a tool called ACR and tested against programs with spatial memory bugs from the SPEC CPU2006 benchmarks and the Software Verification Competition, showing good results but at the cost of a performance overhead.
Bader et al. [24] addresses the issue of automatic fixing common bugs by learning from past fixes. The authors present Getafix, an approach that is able to propose corrections in time proportional to what it would take to obtain static analysis results while producing human-like fixes. Getafix is based on a hierarchical clustering algorithm that summarizes fix patterns into a hierarchy, and it uses a simple ranking technique to select the most appropriate fix for a given bug. The results showed that the tool can perform well and accurately predicts fixes for several bugs, reducing the time developers spend fixing recurring bugs

Chapter 3

Proposed Solution

3.1 Challenges

3.1.1 How to find vulnerabilities by fuzzing without instrumentalizing the application binary?

One challenge that we encounter related to the usage of traditional fuzzers was the fact that the binary of real world production applications, including the ones running in embedded system, are not instrumentalized when compiled, a feature which fuzzers require in order to trigger new internal states to improve the code functional coverage. Moreover, without accessing the application source code, it is not possible to generate an instrumentalized binary. However, our goal is to find vulnerabilities in binary applications without accessing its source code. Our idea to overcome this issue and reach our goal is to use new binary-only fuzzing approaches that have been developed and, at the cost of performance, can perform a fuzzing session on a non-intrumentalized binary. From all those approaches, the one that uses QEMU was chosen, and in addition, it is the one with less performance cost.

3.1.2 How to find a vulnerability sensitive sink without access to the application source code?

One of the challenges of our approach was how to find the vulnerability sensitive sink associated with a buffer overflow, without accessing the source code and only using the non-instrumentalized binary. Our idea to overcome this problem is to use a debugger with recording functionality that enables us not to only follow the expected flow of execution, but also to go back over the flow. This allows us to perform a reverse data-flow analysis from the point that the application crashes (result of a stack overflow vulnerability) to the point that cause that crash, its sensitive sink.

3.1.3 How to remove vulnerabilities by only patching the binary code directly?

One of the main challenges we faced is how to remove a vulnerability by patching the binary code and only have access to it, a requirement not compatible with most common patching techniques. Binary rewriting exists and is a common practice for binary-only applications, but for instrumentation. The current techniques can be either dynamic and static. Dynamic techniques, such as QEMU, requires an external component like an emulator to analyze all interaction with the application at the cost of performance and system requirements. On the other hand, static technique is not feasible for large binaries since the rewriting process will move instructions making it dependent on recovering the control flow information, which is a hard problem by itself.
Because embedded systems are usually extremely optimized and do not have a lot of free resources available, dynamic rewriting was ruled out, leaving static rewriting as the only viable option. But how to overcome its complexity? Our idea to address this issue is to look into instruction punning and the usage of binary trampolines that allow for static binary rewriting to occur without the need for control flow recovery.

3.1.4 How to validate the effectiveness of the patch?

One of the challenges of our approach is how to validate that a patch have successfully mitigated a vulnerability and also, how to ensure that the patch did not break the application logic nor created new vulnerabilities? Our idea to overcome this problem is to add all previously found exploits to our good initial test cases and once again perform a fuzzing session on the patched binary in order to ensure that the already known vulnerabilities have been successfully patched, and no new vulnerabilities have been added.

3.2 Approach Overview

This section describes our proposal for finding and confirming stack overflow vulnerabilities, as well as patching them and validate its effectiveness.
Our approach makes use of a fuzzer on a non-instrumentalized binary in order to find possible vulnerabilities and correspondent exploits. The exploits will then be used to confirm the presence of stack overflow vulnerabilities by means of examining its behaviour and CPU registers in runtime and using a bottom-up data flow analysis approach to determine their sensitive sink. All collected data will then be used to identify the correct patch template on a template database, using it directly on the binary to fix the vulnerability.
The validation process will once again use the exploit(s) that were found to be true positives, to confirm that the patch process was successful, i.e, if the program does not crash or hang anymore, then the patch was successful.

Figure 3.1 illustrates an overview of the approach architecture and its main modules. The approach was designed in such a way that all modules could be used independently, which means the execution flow can also be started from any of the main modules.

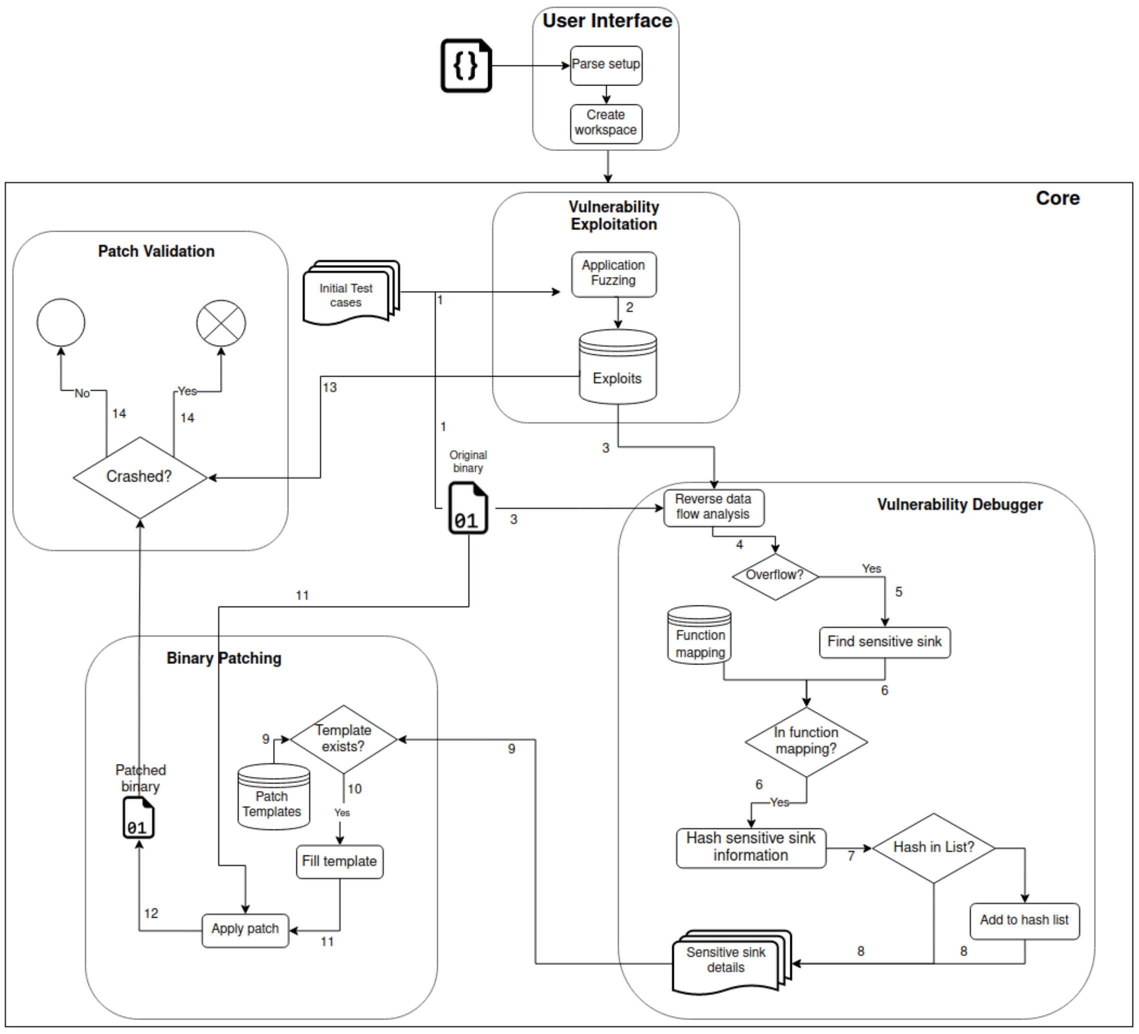


Figure 3.1: Architecture overview

Overall, the approach starts with a ***JSON setup*** file, containing all the setup information needed to run the application, that will be consumed and parsed by the ***User Interface*** module, the start point of our approach. This module will create a workspace for the current session and call the module(s) that match the setup definition. Next, we give an overview of all the components, modules and their interactions.

- ***Original binary* -** This is the original, non-instrumented, binary that we want to test and patch. It will be used in three different modules.

  1. ***Vulnerability Exploitation*** module to find vulnerabilities and their exploits.

2. ***Vulnerability Debugger*** module in order to confirm the vulnerabilities found and get the information of their sensitive sink.
3. ***Binary Patching*** module to patch one or more vulnerabilities found and confirmed and thus generating a new, patched, binary.

- ***Test cases* -** Good examples of an expected input data that would allow the targeted binary to run normally and have a correct and expected behaviour.
- ***Vulnerability Exploitation* -** This module will perform blackbox fuzzing to the given non-instrumentalized binary file, mutating the available test cases in order to generate new ones and thus try to increase the code coverage, and finding the ones that are able to generate a crash/hang.
- ***Exploits* -** This is the database that contains one or more exploits that were either discovered during the fuzzing session or given by the user.
- ***Vulnerability Debugger* -** This module is responsible for confirming that a stack overflow vulnerability occurred, as well as finding its sensitive sink point and detailed information through memory analysis.
- ***Function mapping* -** This is the database that contains detailed information about sensitive sinks, such as the number of arguments, where these are declared in the assembly and how to get their values and sizes.
- ***Patch templates* -** This is the database that contains a set of templates for specific sensitive sinks. These templates contain information about the patch file location as well as its arguments.
- ***Binary patching* -** This module will patch the binary file to remove the vulnerabilities, using for this a static binary re-writer which will generate a new, patched, binary file.
- ***Patch Validation* -** This module will use the patched binary, generated by the binary patching module along with the exploits that were confirmed by the vulnerability debugger, and it will validate if the mitigation process works correctly and the vulnerability was indeed fixed.

3.3 Main Modules

This section describes in details how the main modules in our approach work and how to interact with them.

3.3.1 User Interface

This module acts as the interface with the user, preparing the session workspace and parsing the given setup file to determine the correct execution that follows the options made by the user. It will also determine the binary security, checking for the presence of stack canaries in order to inform the other modules, so they can adjust their execution to these security mechanisms (Figure 3.2)

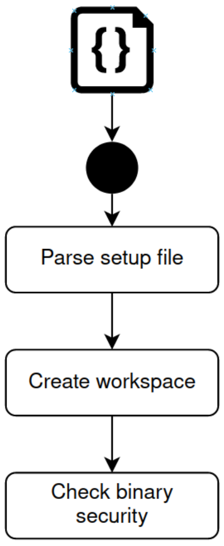


Figure 3.2: User Interface Overview

3.3.2 Vulnerability Exploitation

Our approach main goal relies on the existence of vulnerabilities that can be later confirmed with the exploits that exploited them, and patched accordingly. If there are no known exploits, the Vulnerability Exploitation module (Figure 3.3) can be used to try to find those and pass them along to the following modules.
It starts by receiving a standard binary, without any form of instrumentation, and a set of test cases with expected benign input data that allows the binary to run normally. The module will then prepare a workspace by creating two directories, one for the exploits that will be discovered during the fuzzing session and another for a copy of the initial benign tests cases and where all new test cases generated by the fuzzer will be stored, and with the information provided set all the necessary environment variables that the fuzzer needs to operate. Also, from the setup file, this module will set a target *number of crashes* and a *timeout* that will act as a safeguard to prevent the fuzzing session from running indefinitely.

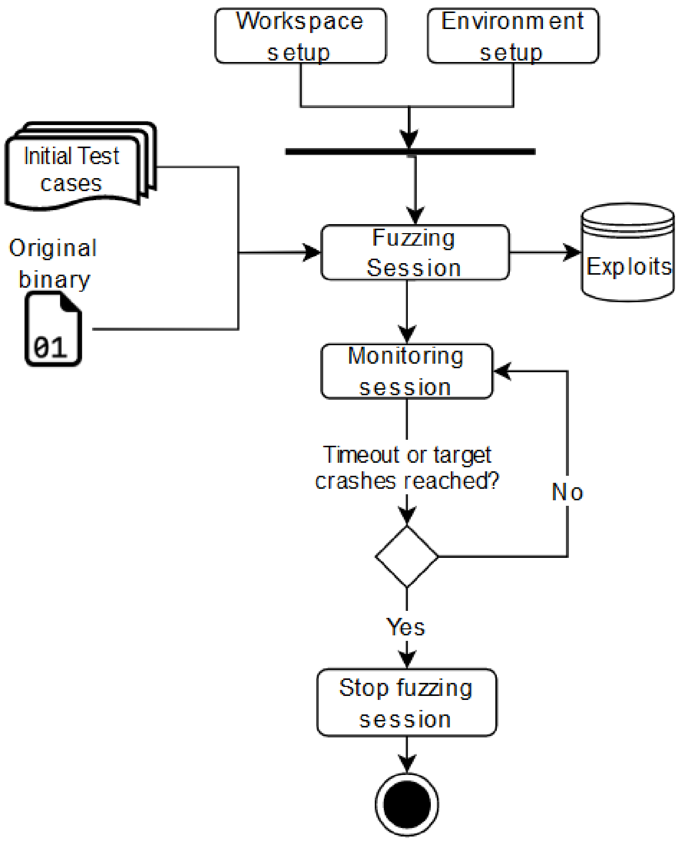


Figure 3.3: Vulnerability Exploitation Overview

Once everything is set up, this module will initiate the fuzzing activity, generating new test cases from the initial test cases and increasing the test coverage in order to find vulnerabilities and saving its exploits. When the fuzzing session starts, this module will monitor its activity in order to stop the current session when the target number of crashes is reached or if there are no new crash discovered during a specific time window, i.e, timeout.

3.3.3 Vulnerability Debugger

One of the main goals of our approach is to confirm and find the sensitive sink of a found vulnerability, collecting as much information as possible about it, such as the number of arguments of a vulnerable call and their size. The Vulnerability Debugger module is the one responsible for these actions, where its workflow is illustrated in Figure 3.4 and Figure 3.5.
The module starts by receiving a list of one or more exploits that crashed the application and so triggering a vulnerability, as well as the vulnerable binary and a set of specifications defined by the user in the setup file that allows this module to know how to interact with the binary, i.e, how to use the exploit inputs.

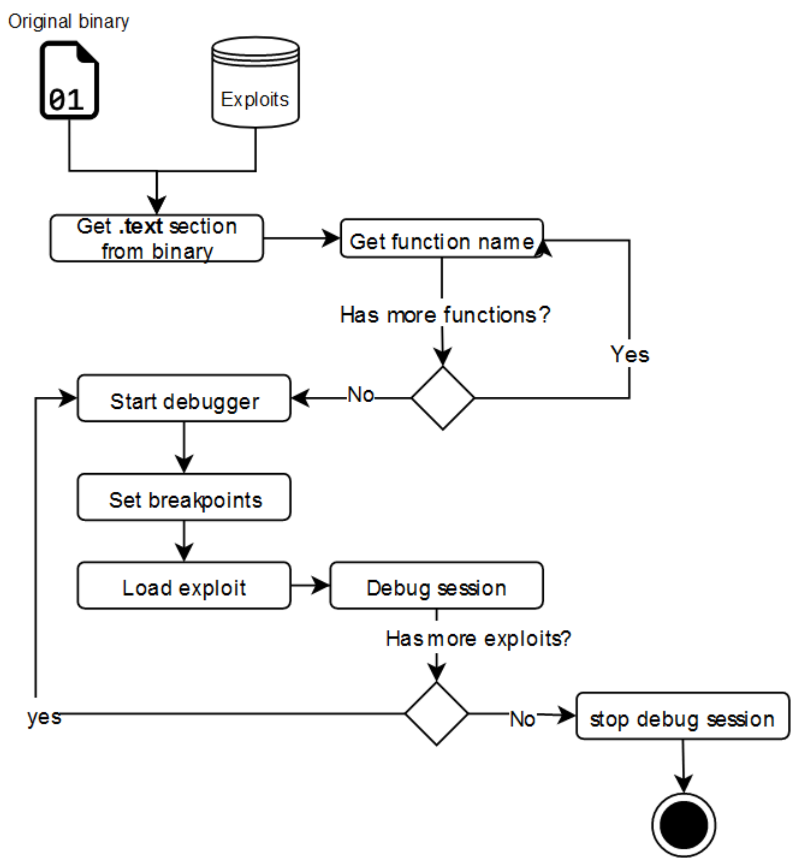


Figure 3.4: Vulnerability Debugger Overview

After loading all the information, it will first extract all the function names from the binary .text section, which contains the program's actual code, and it will open the binary in a debugger, which allows it to closely monitor its execution, setting breakpoints in all the extracted functions and activating the possibility of reverse debugging. Next it will iterate over the exploit list and for each exploit this module will run the application, skipping all breakpoints until it detects a crash, at which at this point it tries to confirm the presence of a stack overflow vulnerability. Figure 3.4 gives an overview of the workflow explained so far, while Figure 3.5 focuses on detail the workflow within the Debug Session.

Because a stack overflow vulnerability occurs when an application fails to bound check a writing operation to the buffer, and the debug session takes place in runtime, the module will have access to the following CPU registers RSP, RBP and RIP. These registers are usually overwritten in this type of vulnerability, therefore, a form to verify the occurrence of such vulnerability is to check their values are pointing to a valid memory address within the current session.

If any value of those registers is found as being invalid, we can confirm that a stack overflow vulnerability occurred. At this point the Vulnerability Debugger module starts the reverse data-flow analysis, using for that the previously set breakpoints as anchors which will allow jumping to the beginning of those functions and re-checking the RSP, RBP and RIP values for each of those functions, until no trace of stack overflow is detected. When the stack overflow vulnerability is no longer detected, the Vulnerability Debugger knows that the vulnerability resides in the current function under analysis, and so it deletes all other previously set breakpoints except the one of the current function. Afterwards, it starts a step-by-step analysis on each assembly line, searching for known vulnerable calls belonging to the Function Mapping database. When it finds such a call, it sets a breakpoint on it and checks the same CPU registers after that call has been executed. If that call triggered a stack overflow, it means the module have found the sensitive sink point, and it can now start the final phase of the debugging session to gather all the information about the vulnerable call.

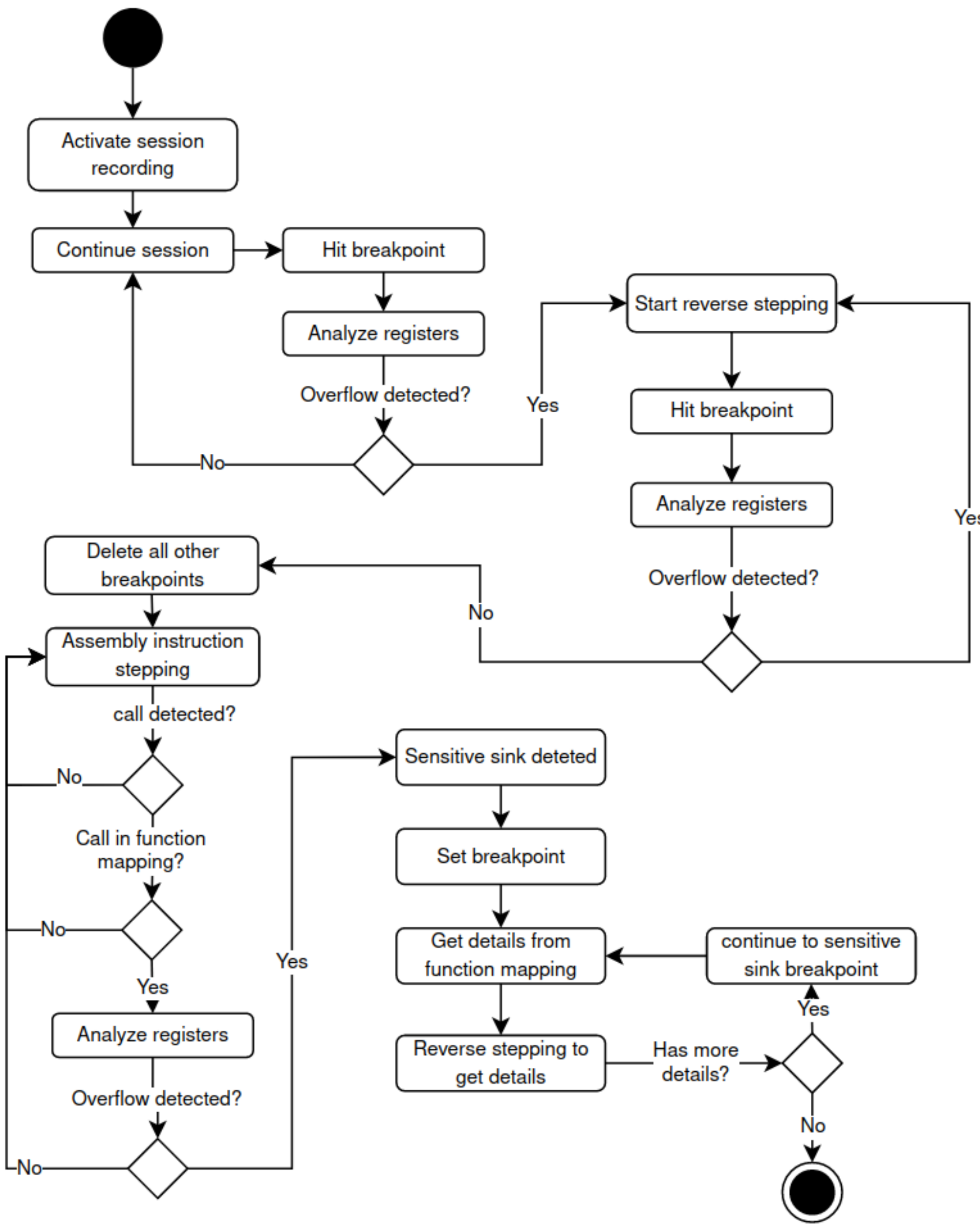


Figure 3.5: Debug Session Overview

At this stage the module will once again delete all breakpoints except the one in the call that triggered the vulnerability, the sensitive sink, checking the Function Mapping database to obtain the details on what information can it retrieve from that call and how can that information be obtained. Afterwards and starting from the sensitive sink call, this module will perform the reverse step-by-step analysis through each assembly line until it is able to obtain the information it needs. When the information is obtained, the module resumes the execution in order to return to the sensitive sink where it had previously set a breakpoint. However, if there is more information to be obtained, it repeats the previous step until no more information is needed (Figure 3.5).

```
Disassembly of section .plt.sec:

00000000000010a0 <strcpy@plt>:
    10a0:       f3 0f 1e fa
    10a4:       f2 ff 25 fd 2e 00 00
    10ab:       0f 1f 44 00 00

00000000000010b0 <puts@plt>:
    10b0:       f3 0f 1e fa
    10b4:       f2 ff 25 f5 2e 00 00
    10bb:       0f 1f 44 00 00

00000000000010c0 <fread@plt>:
    10c0:       f3 0f 1e fa
    10c4:       f2 ff 25 ed 2e 00 00
    10cb:       0f 1f 44 00 00
```

Figure 3.6: Binary PLT section example

When all the information is gathered, this module will search the PLT section of the vulnerable binary to get the offset address of the sensitive sink call. Figure 3.6 shows an example of the PLT section with four functions, and for the function puts, for instance, the offset address it will return would be 0x10b0. Finally, it will compile all the information in a JSON format, for such information to be easily interpreted and consumed by other modules.

3.3.4 Binary Patching

This module receives the vulnerability sensitive sink information, which can be gathered and compiled by the user through the setup file or by the Vulnerability Debugger module. After analyzing the sensitive sink information, this module will extract the vulnerable call name, using it to choose the appropriate patch template from the Patch Template database along with the respective input arguments associated with the sensitive sink and necessary to the patch and then apply the patch to fix the current vulnerability. Each supported vulnerable function call (i.e, a sensitive sink) has at least one patch template, depending on the way the vulnerability is expressed in the code and the information of the sink associated with it. A template is a small pre-compiled C application that receives the CPU registers values as arguments and other relevant information, as the indicative of the presence of stack canaries. The template will be added to the vulnerable binary through the usage of instruction punning, where an *e9 jump* instruction opcode is encoded as a relative jump in order to redirect the application control flow to a trampoline. The trampoline is the area where the patch can run a safe code before returning the control flow to the main application. Figure 3.7 presents a simplistic view of this idea.

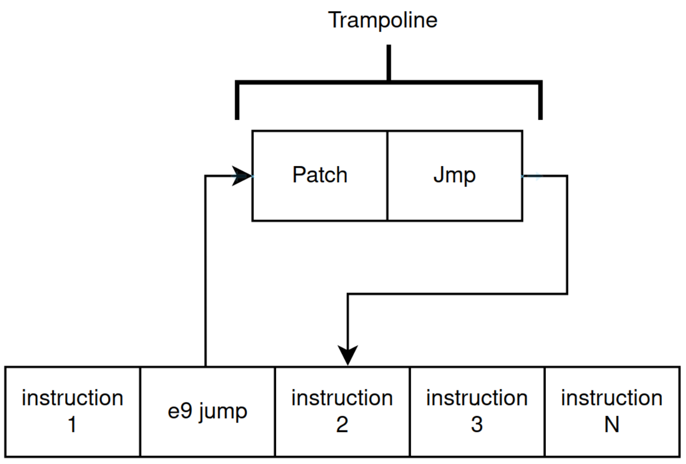


Figure 3.7: Simplistic view of a trampoline

The trampoline is placed at the address where the vulnerable call is located, hence the correct sensitive sink information is crucial and is used to try to pinpoint its location. In our binary patching approach, two types of patches are proposed, one that replaces vulnerable calls with their safe version and another that will run before the vulnerable call, limiting the input maximum size to match the destination buffer length, avoiding thus an overflow.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5
6  void overflow() {
7          char buf[10];
8          printf("User input: \n");
9          gets(buf);
10         printf("User input is %s\n", buf);
11 }
12
13 int main (int argc, char** argv) {
14         overflow();
15         printf("No stack overflow detected!\n");
16         return 0;
17 }
```

Listing 3.1: Vulnerable usage of gets call

For example, Listing 3.1 shows a vulnerability in line 9. Our approach will replace the *gets* function with its safe version, *fgets*. Another example is Listing 3.2 that shows a vulnerability in line 11. Our approach can patch it by either replacing the *strcpy* function with *strncpy*, its safe version, or taking the input that will be copied to the buffer and limit its size if it exceeds the buffer length.

```

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

#define MAXLINE 100


void overflow(char *s) {
        char buf[10];
        strcpy(buf, s);
}

int main (int argc, char** argv) {
        char line[MAXLINE];
        FILE *fp = fopen(argv[1], "r");
        if (!fp){
            fprintf(stderr, "Can't open file\n");
            return 0;
        }

        fread(line, sizeof(line), 1, fp);
        fclose(fp);
        overflow(line);
        printf("No stack overflow detected!\n");
        return 0;
}
```

Listing 3.2: Vulnerable usage of strcpy

Since most safe calls need information about the maximum amount of bytes that can be read or write, the patch can receive this information or calculate it dynamically, depending on the case. One possible and common scenario to obtain this amount is by calculating the distance between the values of RBP and the RDI registers. Listing 3.3 illustrates the *gets* function call, where *0xa* bytes, are allocated from the RBP to the RAX that will be then copied to RDI. In this case, if we calculate the difference between the values of RDI and RBP, we will obtain the 10 bytes buffer size.
If a stack canary exists, we also need to take that into consideration and subtract 8 bytes to the previous calculated size, since the canary will be placed after the RBP.

```
lea      -0xa(%rbp),%rax
mov      %rax,%rdi
mov      $0x0,%eax
callq    1080 <gets@plt>
```

Listing 3.3: Assembly code before the call gets

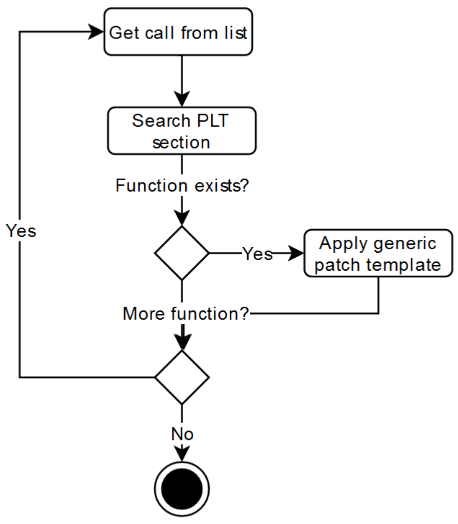


Figure 3.8: Auto-patch overview

Because functions like *scanf* and *gets* have their sensitive sink occurring inside an external library, in this case the glibc, and are almost always vulnerable and should not be used, this module also implements an optional auto-patch functionality that can search the PLT section of the given binary for the presence of these functions, thus gathering their offset address and applying the generic patch template, replaces the vulnerable call with the safe alternative, Figure 3.8 presents an overview of this auto-patch process. These functions can be set by the user in the setup file, along with the ability to turn this feature on or off.

3.3.5 Patch Validation

This module will validate if a patch was successfully applied and will consider that the vulnerability is fixed if none of the exploits from the database, which, were able to exploit the vulnerability and trigger a stack overflow upon execution. Figure 3.9 depicts the workflow of the Patch Validation module.

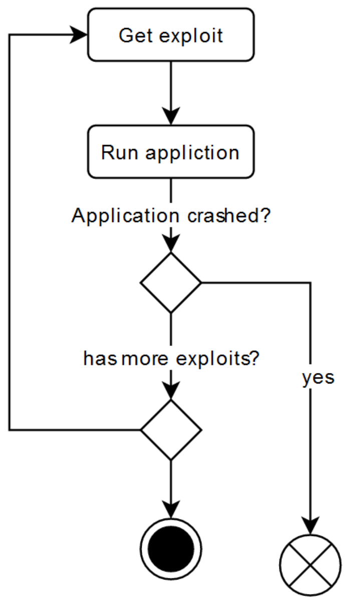


Figure 3.9: Patch validation overview

At a first instance, the use of the exploits can be enough to consider that the patch was successful, but to ensure its validation this module can copy all database's exploits to the initial set of test cases and start the Vulnerability Exploitation module once again in order to initialize a new fuzzing session that will mutate the previously found exploits, trying to find new vulnerable paths and new exploits, i.e, attempt to break the patch or find a new vulnerability that might have been introduced by the patch.

Chapter 4

Implementation

This chapter describes the implementation of the proposed solution presented in Chapter 3. The developed tool was implemented in Python 3.10 and integrates parts of some tools existing in the literature. Section 4.1 describe the tools that were used to build our prototype and Section 4.2 contains detailed information about the implementation of prototype itself.

4.1 Tools used

This section presents the tools that were used in the implementation of some modules of our prototype.

4.1.1 AFL++

AFL++ [32] is considered an evolution of American Fuzzy Lop, AFL [53]. It forked from the original AFL project, incorporating all of its best features, and implementing a lot of optimizations, new and better mutators and source and binary instrumentation modules and new features. This tool was used in the Vulnerability Exploitation module (Section 3.3.2) in order to perform the fuzzing session, using QEMU as the main binary instrumentation. From the binary instrumentation modules available in AFL++, QEMU was chosen because it was the one with the least amount of performance loss when compared with source instrumentation.

4.1.2 GDB

The GNU Debugger (GDB) [35] is one of the most well established debugging tools that allows not only to monitor an application in runtime but also changing its internal state of execution. It was chosen to be used in the Vulnerability Debugger module (Section 3.3.3), and the main reasons to chose GDB were:

1. It supports machine interface (MI), a line based machine oriented text interface intended to allow for a better integration with tools which use the debugger, like the proposed prototype;

2. It allows to record a debugging session, a feature that is extremely important in the context of our approach since it is needed to perform the reverse data flow analysis from the crash point until the sensitive sink.

4.1.3 Pygdbmi

Pygdbmi [12] is a python library that allows to control GDB, as a sub-process, parsing its machine interface output into a structured python dictionary, JSON serializable. This tool was chosen for its programmatic control over GDB, a feature that allows to interact with GDB using any GDB command

4.1.4 E9Tool

E9Tool [7] is a static binary rewriting tool for Linux ELF binaries that uses low level techniques like instruction punning, padding and eviction to replace or insert binary code without having to worry about recovering the control flow information, which is one of the main reasons it was chosen as the patching tool for the proposed solution. Another reason was the fact that E9Tool is highly programmable, intended to be easily integrated into projects and allowing C language to be used to produce patch templates as well as CPU registers to be used and modified in runtime.

4.2 Implementation design

This section presents the implementation design, including the algorithms in pseudocode, with examples of inputs and outputs, that describes the implementation of the modules of the proposed tool, giving thus a better explanation on how it works.

4.2.1 Setup configuration

The setup configuration file illustrated in Listing 4.1 is defined by the user upon starting the tool, containing all the necessary information that it needs to work and allowing the user to fine tune it to its liking.
In the file generic section (lines 2 to 5) the user defines whether the binary was previously instrumentalized or not in order to tool decide if the QEMU module is need or not. Also, he defines the binary path and its input arguments, taking into consideration that the tool will replace the string **%CUSTOM_ARG_HERE%** with the test cases or exploits input. All other sections are directly related to the correspondent module and can be skipped if

the user decides, allowing any module of the tool to be used separately.

```
{
  "is_afl_compiled" : BOOLEAN,
  "binary_path" : "FULL PATH TO BINARY",
  "run_arguments": "%CUSTOM_ARG_HERE%",
  "arg_is_file": BOOLEAN,
  "fuzzing" : {
    "skip_fuzzing": BOOLEAN,
    "test_case_path" : "PATH TO TEST CASES"
  },
  "debugging" : {
    "skip_debugging": BOOLEAN,
    "exploit_path" : "PATH TO EXPLOITS"
  },
  "patching" : {
    "skip_patching": BOOLEAN,
    "template_root": "PATH TO PATCH TEMPLATES",
    "call_name": "CALL NAME",
    "function_name": "FUNCTION NAME",
    "max_size": SIZE,
    "real_address": "ADDRESS IN THE PLT SECTION",
    "has_canary": BOOLEAN
  },
  "auto_patching" : {
    "skip_auto_patching": BOOLEAN,
    "auto_patch_functions": ["gets", "scanf"]
  },
  "validate": {
    "skip_patching": BOOLEAN,
    "exploit_path" : "PATH TO EXPLOITS",
    "patched_binary_path" : "PATH TO PATCHED BINARY"
  }
}
```

Listing 4.1: Setup configuration file example

After the user interface module digests and parses the configuration file, it will determine the correct order in which the modules need to be executed, and it will create the workspace for the current session with the name of the binary file concatenated with a random string, allowing this way the creation of multiple sessions for the same binary without overwriting any previous session.

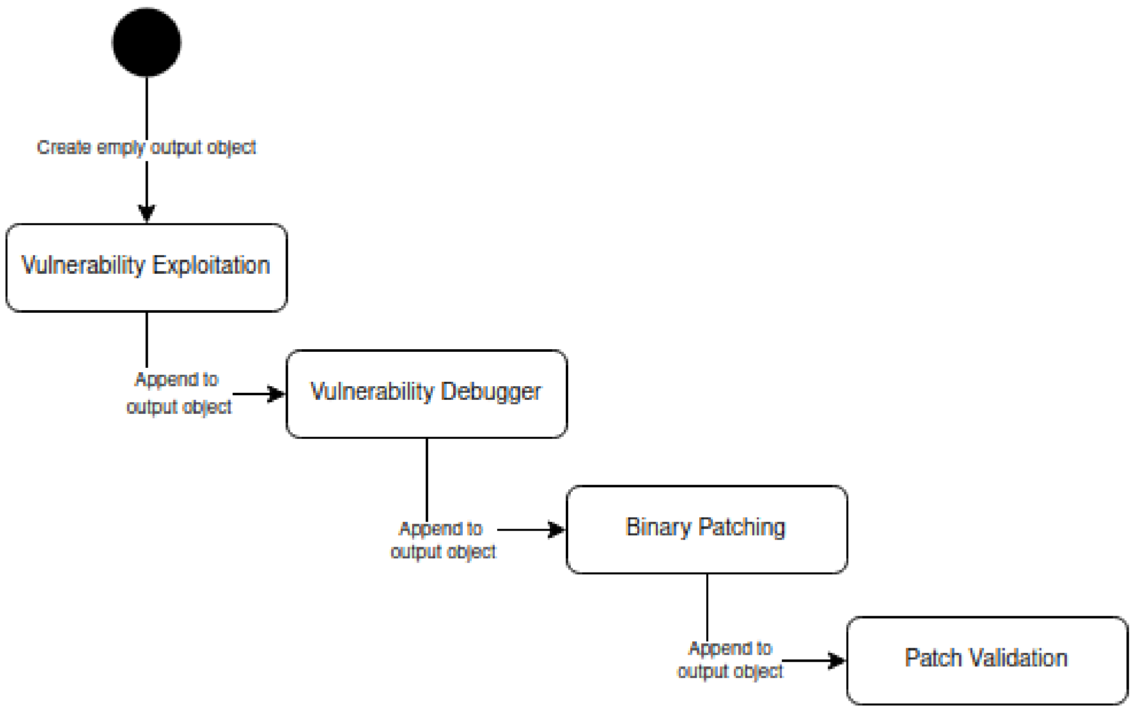


Figure 4.1: Natural execution order

The natural order of execution, if no modules are skipped, is the one described in Figure 4.1. In the beginning of the execution, an empty object is created, so each module can later add information to it, which will be used as the input for the following modules. If a module is skipped, the information in the configuration file of the next module to be run will be appended by the User Interface module before it is called.
Once the fuzzing session finishes, this module will append to the output object all information related to the session, such as the exploit location path and the number of exploits that were found.

4.2.2 Vulnerability Exploitation

The Algorithm 1 is responsible for executing a fuzzing session to a given binary. It starts by receiving, from the configuration file, the values of the *timeout* and *target number of crashes* for the current session, as well as the paths for the binary, test cases and exploit directory where the exploits are going to be saved. After everything is set, the fuzzing session starts along with a monitoring process that will constantly check the session in order to determine whether the target number of crashes or the defined timeout was reached, and if so the fuzzing session is stopped

Algorithm 1 Vulnerability exploitation algorithm

```
Input:
    $timeout > 0$
    $num\_of\_crashes > 0$
    $binary\_path \neq$ ""
    $test\_cases \neq \{\}$
    $exploit\_path$ directory is empty
Output:
    $exploit\_list$

 1: $pid \leftarrow start\_fuzzing(binary\_path, test\_cases, exploit\_path)$
 2: $monitor \leftarrow start\_monitor(pid)$
 3: $stop \leftarrow false$
 4: while $stop$ is false do
 5:     $num\_of\_crashes \leftarrow get\_current\_crashes(monitor)$
 6:     $timeout \leftarrow time\_since\_last\_crash(monitor)$
 7:     if $num\_of\_crashes$ is reached OR $timeout < time\_since\_last\_crash(monitor)$
    then
 8:         $stop \leftarrow true$
 9:     end if
10: end while
```

4.2.3 Exploit Iteration

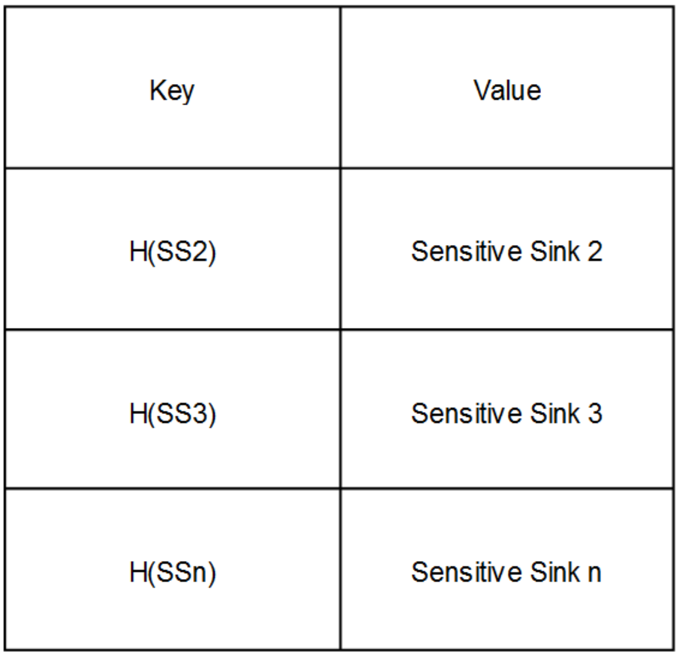
The main goal of the exploit iteration (Algorithm 2) is to iterate over a list of raw exploits that were either gathered by the Vulnerability Exploitation module (Section 3.3.2) or provided by the user, calling the debug session for each of these in order to confirm the vulnerability and collect its sensitive sink.

Algorithm 2 Vulnerability exploitation algorithm

```
Input:
    $raw\_exploit\_list \neq \{\}$
    $binary\_path \neq$ ""
Output:
    $exploit\_dict$

1: for each $exploit \in raw\_exploit\_list$ do
2:     $sensitive\_sink \leftarrow debug\_session(exploit, binary\_path)$
3:     if $sensitive\_sink \neq \emptyset$ AND $hashed(sensitive\_sink) \notin keys(exploit\_dict)$
   then
4:         $exploit\_dict[hashed(sensitive\_sink), sensitive\_sink]$
5:     end if
6: end for
```

Because different exploits can trigger the same vulnerability and thus have the same sensitive sink information, after a debug session was successfully executed and returned, this algorithm will calculate a hash composed by the sensitive sink information which will be compared with all the existing keys in the exploit dictionary structure, as shown in Figure 4.2. If no match found, a new vulnerability is detected, and so the sensitive sink information will be added to the exploit dictionary along with its hash as the key.



Key	Value
H(SS2)	Sensitive Sink 2
H(SS3)	Sensitive Sink 3
H(SSn)	Sensitive Sink n

Figure 4.2: Exploit dictionary structure

4.2.4 Stack overflow confirmation

The Algorithm 3 illustrates how a stack overflow vulnerability is confirmed by the vulnerability debugger module.
Advanced Vector Extensions 2 (AVX2), a vectorization extension that expands most integer commands to 256 bits and introduces new instructions, is used by most modern CPUs but is not yet fully supported by GDB recording feature due to it being poorly maintained. Because of this and to avoid the debug session to fail, specially on Intel CPUs, before the algorithm starts AVX2 support needs to be turned off by using the GDB flags *glibc.cpu.hwcaps=-AVX2_Usable,-AVX_Fast_Unaligned_Load*. Afterwards, the algorithm starts by getting all function names and the places they reside in the binary file, issuing for this the system command *objdump* to look for them at the *Procedure Linkage Table* (PLT) section. Next, it uses the places of the functions to set break points, so it is easier to jump between functions during the debugging session.

Algorithm 3 Finding vulnerability crash point

Input:
binary_path

Output:
crash point

```
1: $function\_names \leftarrow get\_functions\_from\_binary(binary\_path)$
2: $gdb \leftarrow start\_gdb\_session(binary\_path)$
3: for each $function \in function\_names$ do
4:     $gdb.write$(”break $function$”)
5: end for
6: $gdb.run(exploit)$
7: $gdb.write$(”record”)
8: while $detected\_crash$() is false do
9:     $gdb.continue$()
10: end while
11:
12: if $detected\_overflow$(RIP, RBP, RSP) is true then
13:     continue                                  ▷ Stack overflow confirmed
14: else
15:     breaks
16: end if
```

At this point, everything is set, and the debug session is initialized with the potential exploit as input. Because break points were set for all functions, after the session starts the algorithm will hit the first breakpoint (usually the main function) and a record instruction is issued so GDB starts recording the current session and enables the reverse data flow analysis. It will then continue, skipping all breakpoints until it detects a crash, i.e, a segmentation fault, which indicates that the exploit input have worked as expected.

At this point, because we are monitoring the application in runtime, the algorithm issues several GDB commands to get the CPU registers *RIP*, *RBP* and *RSP* where their values are smashed when a stack overflow vulnerability occurs, and so the algorithm will evaluate these in order to assess if they are invalid or point to an invalid memory address, confirming the vulnerability.

4.2.5 Finding Sensitive Sinks

The Algorithm 4 demonstrates how a vulnerability sensitive sink point is found after a stack overflow is confirmed (Section 4.2.4).
Having the current debug session recorded, and break points set in all functions, this algorithm, from the crash point, will start a loop by issuing the command *reverse continue*

that allows to jump backwards to the previous called functions and get the values of the CPU registers, to be analyzed to confirm if the vulnerability is still detected.

Algorithm 4 Find sensitive sink

Input:
stack overflow is confirmed
Output:
sensitive sink information

```
1: while $detected\_overflow$(RIP, RBP, RSP) is true do
2:     $gdb.write$("reverse continue")
3: end while
4:                                   ▷ Found function with sensitive sink.
5: while $detected\_overflow$(RIP, RBP, RSP) is false do
6:     $gdb.write$("next instruction")
7: end while
8:
9: $gdb.write$("delete breakpoints")
10: $gdb.write$("break here")                    ▷ Found sensitive sink.
```

If the vulnerability is still detected, the loop continues, otherwise it stops because it found the last known function before the vulnerability has occurred, which means this function is most likely to be the one where the sensitive sink will be found. At this point, with the exception of the current breakpoint, all existing breakpoints are deleted, and the algorithm shifts its focus to the current function, stepping into it and iterating over all its assembly instructions, setting break points whenever a call is identified and analyzing the CPU registers after the call is executed to detect an overflow.
When an overflow is detected, it means the algorithm found the sensitive sink. At this point, it will delete all other breakpoint except the one associated with the call that have originated the vulnerability. This breakpoint is used by the algorithm as a hook, allowing it to quickly return to this point when it needs to.

4.2.6 Getting call details

Once the sensitive sink point is discovered, the Algorithm 5 starts to look in the assembly code for the current C library function call and, using an auxiliary function mapping database, determines how to obtain all possible details about the function itself. This database contains information about the following vulnerable function calls - *gets*, *strcpy*, *strcat*, *sprintf* or *scanf*. The function mapping database, represented in Listing 4.2, is a JSON structure that contains information and instructions for each argument of a given function call.

Algorithm 5 Getting sensitive sink call details

Input:
sensitive sink is confirmed

Output:
sensitive sink details

```
1: $call\_name \leftarrow gdb.write$(”get call name”)
2: $argument\_instructions \leftarrow get\_call\_instructions(call\_name)$
3: for each $instruction \in argument\_instructions$ do
4:     $steps \leftarrow instruction.get\_step\_number()$
5:     $gdb.write$(”reverse $steps$ instructions”)
6:     $info \leftarrow gdb.write$(”get information”)
7:     $sensitive\_sink$ append $info$
8:     $gdb.write$(”continue”)                    ▷ go back to sensitive sink
9: end for
```

The information in the database includes:

- **The argument type**. This will determine what information will be retrieved and how it will be retrieved, i.e, if type *string*, the Algorithm will get its value.
- **Number of reverse steps needed**, from the sensitive sink, to get access to the argument
- **The position this argument takes in the call**. This is important so during patching we know what register we will need to modify, i.e. if it is first position, we need to modify RDI (RDI is the register that contains the first argument).
- If this argument is (or not) the cause of the vulnerability.

```
"arguments": [
    {
        "info": "src",
        "line": "12",
        "name": "s",
        "size": 24,
        "taintable": "True"
    },
    {
        "info": "dst",
        "line": "12",
        "name": "buf",
        "size": 10,
        "taintable": "False"
    }
],
"call_name": "strcpy",
"overflow_detected": true,
"sensitive_sync": "=> 0x5555555551c5 <test+30>:
    callq  0x555555555070 <strcpy@plt>",
```

Figure 4.3: Sensitive sink information output

For each argument the algorithm will, from the sensitive sink, reverse the number of steps it retrieved from the function mapping and according to the argument type it will either get its value or allocated size.
Once everything is done, it compiles all the gathered information and generates a JSON formatted string as output (Figure 4.3) that can be easily consumed by other modules as well as interpreted by a human.

```
{
"strcpy":{
        "args":{
            "dst":{
                "type":"buffer",
                "reverse_steps":3,
                "call_pos": 1,
                "taintable":true
            },
            "src":{
                "type":"string",
                "reverse_steps":4,
                "call_pos": 2,
                "taintable":false
            }
        }
    }
}
```

Listing 4.2: Function mapping example for *strcpy* function

4.2.7 Patching a binary

The Algorithm 6 details how the patching process works from the moment it receives a sensitive sink object, containing all previously collected information about a vulnerability, to the moment a patch is issued using the *e9patch* command from the *e9tool* toolkit.

Algorithm 6 Get sensitive sink call details

```
Input:
    $sensitive\_sink \neq \{\}$
Output:
    patched binary

1: $call\_name \leftarrow sensitive\_sink$.get_call()
2: if get_template($call\_name$) exists then
3:     $template \leftarrow$ get_template($call\_name$)
4:     fill template
5:     issue 9epatch command
6: end if
```

First, it will get the patch template, from the database, that is associated with the sensitive sink function name. A patch template is a JSON a structure (Listing 4.3) containing an *e9patch command string* with certain flags (easy to spot since they are all capital letters) that need to be replaced by the data collected from the sensitive sink in order to call a compiled patch for the given call, which was generated previously and automatically. The data needed to be replaced in a patch may vary from template to template, and it mostly depends on the arguments that the patch requires to work properly.

```
{
    "strcpy":{
        "command":"replace apply_patch(\"{HAS_CANARY}\", RBP,
            RDI, RSI)@{TEMPLATES_ROOT}/strcpy_patch"
    }
}
```

Listing 4.3: patch template for strcpy function

The command itself can be interpreted has:
[REPLACE|BEFORE|AFTER]$FUNCTION\_EXECUTE(ARG1, ..., ARGN)$
$@PATH\_TO\_PATCH\_BINARY$

When the placeholders in the template are filled with the collected data, the patch template is converted into a system command in order for the *e9patch* to be invoked to patch the vulnerable binary, creating a new binary file with the same name and with the word PATCHED appended to it.

4.2.8 Patch

A patch is a compiled C file that respects the *e9tool* syntax [3], with one or more functions that are used to patch a given vulnerability (Listing 4.4).

```
#include "stdlib.c"
#include <stdio.h>

void apply_patch(char *has_canary, void *rbp, void *rdi, char
    *rsi){
    long offset = 0;
        if(strcmp("True", has_canary) == 0){
                offset = 8;
        }
        long size = (long)rbp - (long)rdi - offset;
        strncpy(rdi, rsi, size-1);
}
```

Listing 4.4: strcpy patch example

In our solution, the function to be called is always named **apply_patch**, and it receives information about the existence or not of a canary, as well as the specific register values that depend on the call that is being patched. In our solution, the patch tries to calculate the buffer size given the offset of the specific register to the RBP, using it to define the maximum allowed buffer size. It will then replace the vulnerable call with the safer alternative, keeping the execution flow as close to the original as possible. Full list of patch templates available in Appendix A

Chapter 5

Evaluation

This chapter describes the evaluation process that was performed to assess our tool and discusses its results. Section 5.1 details the evaluation setup and the metrics that were used to evaluate the tool. Sections 5.2 and 5.3 detail the evaluation with synthetic datasets and real applications, respectively, both comprising C programs and compiled with and without necessary protection, i.e, canaries.
Considering the challenges identified in Section 3.1 and their respective solutions proposed to solve them, a set of key elements were needed to evaluate these tool: the ability to find the sensitive sink, the ability to fix vulnerabilities in a binary and the effectiveness of such fixes.Based on this key elements, we defined the following questions:

Q1: Can the tool detect a sensitive sink given an exploit?

Q2: Is the tool capable of fixing a vulnerability in binaries with and without security necessary protections?

Q3: Does the patch introduce new vulnerabilities?

Q4: How does the patch affect the overall performance?

Q5: By how much does the patch increase the final size of the binary?

Q6: Is the tool capable of debugging and fixing vulnerabilities in real applications?

In order to answer these questions, the evaluation process was conducted, and its details are described in the upcoming sections.

5.1 Evaluation setup

To better evaluate our tool, the evaluation process was divided in two parts. In the first part, we used a synthetic dataset of small vulnerable C programs (some containing very few code execution paths), either designed specifically to test this project or downloaded

from Software Assurance Reference Dataset (SARD) [11]. For the second part, we used real applications, which we modified in order to add a vulnerable call that would allow us to test our tool in a scenario with applications containing multiple code paths. Our modifications were very minor, where we either replaced a safe call for its vulnerable alternative or we added a vulnerable call before an output was returned.
All the programs were compiled twice, with and without security necessary protection canaries, so we could evaluate the effectiveness of the patching module and the patch templates we created for this tool. In order to test the performance overhead introduced by the patch, we designed a small script, illustrated in Listing 5.1, that will run both the patched and the vulnerable binaries 5 times with valid, good inputs, that would not trigger an overflow (otherwise the vulnerable binary would crash). For each binary, we collected the execution time, and measured the average time of the 5 runs.

```
patched_bin_path = "/PATH/TO/PATCHED/BINARY"
bin_path = "/PATH/TO/VULNERABLE/BINARY"
input_path = "/PATH/TO/INPUT/FILE"
// True if the binary receives the input from stdin
need_stdin = False

for i in range(0, 5):
    print("Running %d" % (i+1))
    original_time = 0
    patched_time = 0
    if need_stdin:
        original_time = run_test_with_stdin(bin_path)
        time.sleep(1)
        patched_time = run_test_with_stdin(patched_bin_path)
    else:
        original_time = run_test(bin_path, input_path)
        time.sleep(1)
        patched_time = run_test(patched_bin_path, input_path)


    original_bin_runtime.append(original_time)
    patched_bin_runtime.append(patched_time)

average_runtime_original = sum(original_bin_runtime)/
                               len(original_bin_runtime)
average_runtime_patched = sum(patched_bin_runtime)/
                               len(patched_bin_runtime)
average_overhead_percentage = (average_runtime_patched/
                               average_runtime_original)*100
```

Listing 5.1: Performance script snippet

5.2 Evaluation with synthetic applications

We created 10 applications for testing purposes, namely two for each vulnerable supported call, and we tried to cover the most common programming practices seen by real applications. To increase the dataset of synthetic applications, we added 8 applications from SARD database that contained at least one vulnerable supported call. Table 5.1 shows the total amount of synthetic applications, downloaded from SARD or created, that were used during this evaluation process.

Table 5.1: Summary of synthetic test cases from SARD and created for testing purposes

Vulnerable functions	Num. of applications
gets	3
strcpy	4
strcat	5
sprintf	3
scanf	3

The average testing results, grouped by vulnerable function, of all synthetic applications, compiled without canaries are shown in Table 5.2. Table 5.3 shows the results for these applications but compiled with canaries.

Table 5.2: Average results for synthetic applications without canaries

	gets	strcpy	strcat	sprintf	scanf
Debug successful?	No	Yes	Yes	Yes	No
Debug time (s)	N/A	141	185	128	N/A
Patch successful?	Yes	Yes	Yes	Yes	Yes
% File size increase	182.79%	181.39%	180.47%	181.60%	182.65%
% Execution time overhead	26.36%	44.83%	38.59%	35.34%	16.01%
Patch validated successfully?	Yes	Yes	Yes	Yes	Yes

Table 5.3: Average results for synthetic applications with canaries

	gets	strcpy	strcat	sprintf	scanf
Debug successful?	No	Yes	Yes	Yes	No
Debug time (s)	N/A	146	192	147	N/A
Patch successful?	Yes	Yes	Yes	Yes	Yes
% File size increase	181.03%	178.46%	178.29%	178.46%	181.71%
% Execution time overhead	28.76%	57.12%	42.64%	51.55%	30.86%
Patch validated successfully?	Yes	Yes	Yes	Yes	Yes

Based on the results above we can see that only the vulnerable calls *gets* and *scanf* were not able to be processed by the Vulnerability Debug module because the vulnera-

bility takes place inside the *glibc* library where the user input is requested triggering the overflow, and this module does not consider these type of system libraries for debugging. All the other vulnerable calls were successfully debugged their sensitive sink found, and on average the ones that were compiled with canaries took more time to be debugged than the ones that were not. This answers the question **Q1** affirmatively, except for the cases where the sensitive sink takes place inside an external library, i.e, *glibc*. Therefore, the tool is capable of detecting the vulnerable calls associated with buffer overflows.
Regarding the Binary Patch module, all applications, with and without canaries, were patched successfully with a patch template that replaced the vulnerable call for its safe alternative. Based on this information, we can answer affirmatively to the question **Q2**. All patches produced a binary with 47576 bytes, regardless of the sensitive sink or the binary initial size. This is due to the way that *e9patch* allocates space for the trampolines, in which it divides the virtual address space into sets of blocks with fixed length. In fact, this is why the file size increase percentage is less on the binaries with canaries than the ones without canaries, since the former have a greater initial size due to the canary information. These facts and observations allows us to answer the question **Q5**.
We also analyzed the patched binaries runtime performance by measuring the overhead when compared to the original binary. We found that on average, each patched binary took around 40% more time to execute. This happens even when a binary have the same vulnerable call being patched in multiple locations, where for each vulnerability we notice an increase of around 13% in runtime performance as illustrated in Table 5.4. This information answers the question **Q4**

Table 5.4: Performance overhead of multiple vulnerable calls being patched (without canaries)

	strcpy	
	one vulnerable call	two vulnerable calls
% Execution time overhead	44.80%	57.94%

All the patched binaries were successfully validated, by adding the previously found exploits to the current test case database and letting the Vulnerability Exploitation module run for 6 hours without any vulnerability being found. Base on this, we can answer the question **Q3**, stating that the applied patches did not introduce any new vulnerability.

5.3 Evaluation with Real Applications

We downloaded two applications from GitHub, so we could test our tool on a more robust environment. Since we need specific vulnerable functions to be available in the application and for testing both the Vulnerability Debugger module and the Binary Patching module, we decided to modify these applications. To do so, we replaced the safe functions they contained with their vulnerable alternative, creating thus new sensitive sinks that could be exploited, debugged and patched by our tool. Table 5.5 gives an overview of these modifications

Table 5.5: Summary of modifications introduced in the real applications

Application	Modifications
TicTacToe [18]	Replaced fgets with gets (line 33 of main.c)
Simple Password Generator [15]	Added strcpy before the return instruction (line 105)

Details about the modified source code for the TicTacToe application is showed in Listing 5.2 and for the Simple Password Generator in Listing 5.3.

```
printf("\n(turn #%i) To which square
   would you (player %c) like to move? ", turn, player);

gets(input); // original was fgets(input, 3, stdin)
moveTo = atoi(input);

if(mv(board, moveTo, player))
   turn++;
```

Listing 5.2: TicTacToe added vulnerability

```
pw = gen_pw(length);
char buf[10]; // added
strcpy(buf, pw);    //added
printf("%s\n",buf); //added
return 0;
```

Listing 5.3: Simple Password Generator added vulnerability

Both applications were compiled with and without canaries, and all tests followed the same methodology as the ones in Section 5.2.

Table 5.6: Test results for real applications without canaries

	TicTacToe	Simple Password Generator
Lines of Code	126	160
Debug successful?	No	Yes
Debug time (s)	N/A	6191
Patch successful?	Yes	Yes
% File size increase	176.48%	179.60%
% Execution time overhead	46.50%	62.36%
Patch validated successfully?	Yes	Yes

The results presented in Table 5.6 are very similar to the ones obtained with synthetic applications. All applications were successfully patched and validated, answering the question **Q6** and demonstrating the ability of our tool on discovering and patching vulnerabilities even in more robust environments.

It is important to note that the Simple Password Generator application took more than one hour to be debugged successfully in order to find the sensitive sink due to the way it is programmed. Before the vulnerability is triggered, the application has to iterate over a given length value, generating each character, one by one, and ensuring that the security best practices are followed, as illustrated in the code snippet in Listing 5.4. Because the initial length value was 35 and each step of this iteration (including all security validations) had to be analyzed by the Vulnerability Debugger module, it took more time to find the vulnerability sensitive sink when compared with the other applications under analysis.

```
for (i = 0; i < length; i++) {
    nrand = rand() % 100;
    if (nrand > 50)
        pw[i] = s_latin[rand() % sizeof(s_latin)];
    else
        if (nrand > 20 && nrand <= 50)
            pw[i] = s_glatin[rand() % sizeof(s_glatin)];
        else
            pw[i] = s_esc[rand() % sizeof(s_esc)];
}
return pw;
```

Listing 5.4: Simple Password Generator snippet

Chapter 6

Conclusion

In this dissertation we focused on the problem of buffer overflow vulnerabilities in C, and proposed a solution to mitigate them, by repairing programs with such vulnerabilities. To pursuit with this goal, we analyzed some examples of stack overflow vulnerabilities in C, and we have also examined some functions that are considered insecure for this class of vulnerabilities and their secure versions. In addition, we analyzed several tools and previous work made on the topic of vulnerability detection and automatic software repair, without requiring access to the source code. Our analysis showed that modern fuzzers already support binary-only fuzzing, allowing for non-instrumentalized binaries to be fuzzed at the cost of performance. The analysis showed that some existing debuggers, with the ability of recording a session, could be used on a binary along with a possible exploit, to perform a reverse data-flow analysis to confirm the presence of a stack overflow vulnerability and find its sensitive sink. In addition, we have concluded that some binary rewriting tools, mainly used for binary instrumentation in software functionality testing, could also be used in software security, to patch a binary vulnerability without having access to the program source code. Thus, we proposed a fully automated tool, capable of detecting, confirming and patching vulnerabilities directly on the binary code without it being instrumentalized and without accessing to the source code of the program.
We implemented a prototype of the proposed solution by the combination of three open-source tools, AFL++, GDB and E9Patch, which were the key elements of the implementation of three modules, Vulnerability Exploitation, Vulnerability Debugger and Binary Patching.
The prototype was evaluated using a dataset of vulnerable applications collected from SARD and real applications collected from GitHub. The experimental results showed that the tool was able to successfully detect and confirm vulnerabilities related with buffer overflows, with the exception of those whose vulnerability happens inside a system library, i.e, *glibc*. The results also showed that all vulnerabilities were successfully patched, even those associated with system libraries, and that the tool was able to operate on binaries that were compiled with or without security mechanisms, as the canaries.

Based on these results, we conclude that our solution satisfies the objectives proposed for this thesis and can be a valuable asset for work related to binary patching.

6.1 Future Work

In this section, we present some aspects that could be used to improve the tool and future directions for continuing this work.
GDB record feature proved to be very immature and so, one of the improvements that can be made is replacing GDB with Mozilla Record&Replay tool [14]. This last executes an application and collects traces of its execution, allowing it to be replayed in GDB and fully supporting AVX. Since it uses GDB for its replay feature, it can be seamlessly integrated in our Vulnerability Debug module. Another aspect that can be improved is the creation of more patch templates, for covering more functions associated with buffer overflows vulnerabilities.
For future work, we believe that more vulnerable functions should be mapped, and patch templates created in order for the tool to be able to fix more vulnerabilities. We also believe that in the future, this tool should also consider heap overflows vulnerabilities.

Appendix A

Patch Templates

A.1 Strcpy With size

```c
#include ”stdlib.c”
#include <stdio.h>

void apply_patch(char *has_canary, void *rbp, void *rdi, char *rsi,
    long offset = 0;
        strncpy(rdi, rsi, size-1);
}
```

A.2 Strcpy truncate rsi

```c
#include ”stdlib.c”
#include <stdio.h>

void apply_patch(char *has_canary, uint *rbp, uint *rdi, char *rsi){
    long offset = 0;
        if(strcmp(”True”, has_canary) == 0){
                offset = 8;
        }
        long size = (long)rbp - (long)rdi - offset;

    if(strlen(rsi) > size){
                char *new_rsi = ””;
                strncpy(new_rsi, rsi, size-1);
                new_rsi[size-1] = 0;
                *rsi = *new_rsi;
        }

}
```

A.3 Strcat

```
#include "stdlib.c"
#include <stdarg.h>
#include <stdio.h>

void apply_patch(char *has_canary, void  *rbp, void *rdi, void *rsi)
        long offset = 0;
        if(strcmp("True", has_canary) == 0){
                offset = 8;
        }

        // strlen(rdi) to account for the fact
    // the initial buffer might already have content
        long size = (long)rbp - (long)rdi - offset;
        strncat(rdi, rsi, size - strlen(rdi) - 1);
}
```

A.4 Scanf

```
#include "stdlib.c"
#include <stdarg.h>
#include <stdio.h>
#define MAX 255

void apply_patch(char *has_canary, void  *rbp, void *rdi, void *rsi,
        long offset = 0;
        if(strcmp("True", has_canary) == 0){
                offset = 8;
        }

        long size = (long)rbp - (long)rsi - offset;
        char *buf[size];
        fgets(buf, size-1, stdin);

        va_list aptr;
        va_start(aptr, rsi);
        snprintf(rsi, size-1, rdi, buf);
        va_end(aptr);

        va_end(aptr);
}
```

A.5 Gets

```c
#include "stdlib.c"
#include <stdio.h>

void apply_patch(char *has_canary, uint *rbp, uint *rdi){
    long offset = 0;
        if(strcmp("True", has_canary) == 0){
                offset = 8;
        }
        long size = (long)rbp - (long)rdi - offset;
        fgets(rdi, size, stdin);
}
```

A.6 Sprintf

```c
#include "stdlib.c"
#include <stdarg.h>
#include <stdio.h>
#define MAX 255

void apply_patch(char *has_canary, void  *rbp, void *rdi, void *rsi,
        long offset = 0;
        if(strcmp("True", has_canary) == 0){
                offset = 8;
        }

        long size = (long)rbp - (long)rdi - offset;

        va_list aptr;
        va_start(aptr, rsi);
        vsnprintf(rdi, size-1, rsi, aptr);
        va_end(aptr);

}
```

Bibliography

[1] Cve. https://www.cve.org/About/Overview [Accessed on 14.12.2021].

[2] Data execution prevention. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention [Accessed on 03.03.2022].

[3] E9tool user-guide. https://github.com/GJDuck/e9patch/blob/master/doc/e9tool-user-guide.md2-patch-language [Accessed on 21.03.2022].

[4] Format of executable binary files. https://pierrelib.pagesperso-orange.fr/exec_formats/a.out_freebsd.html [Accessed on 12.06.2022].

[5] Gdb - process record. https://sourceware.org/gdb/wiki/ProcessRecord [Accessed on 18.12.2021],.

[6] Gdb reverse debug. https://sourceware.org/gdb/wiki/ReverseDebug [Accessed on 18.12.2021].

[7] Gjduck/e9patch: A powerful static binary rewriting tool. https://github.com/GJDuck/e9patch [Accessed on 15.04.2022].

[8] Ibm - what is industry 4.0. https://www.ibm.com/topics/industry-4-0 [Accessed on 17.01.2022].

[9] Intel® inspector. https://www.intel.com/content/www/us/en/developer/tools/oneapi/inspector.html [Accessed on 02.01.2022].

[10] Most secure programming languages. https://www.whitesourcesoftware.com/resources/research-reports/what-are-the-most-secure-programming-languages/ [Accessed on 18.12.2021].

[11] Nist software assurance reference dataset. https://samate.nist.gov/SARD/ [Accessed on 07.01.2022].

[12] Pygdbmi. https://cs01.github.io/pygdbmi/ [Accessed on 21.03.2022].

[13] Radamsa. https://gitlab.com/akihe/radamsa [Accessed on 07.01.2022].

[14] Record&replay - lightweight recording deterministic debugging. https://rr-project.org/ [Accessed on 21.05.2022].

[15] Simple password generator. https://github.com/codeliveru/pgen [Accessed on 07.04.2022].

[16] Sonarqube. https://www.sonarqube.org/ [Accessed on 02.01.2022].

[17] Threatpost - vulnerable cardiac devices. https://threatpost.com/st-jude-medical-patches-vulnerable-cardiac-devices/122955/ [Accessed on 25.01.2022].

[18] Tictactoe. https://github.com/emacdona/tictactoe [Accessed on 07.04.2022].

[19] Wired - jeep remote vulnerability. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/ [Accessed on 25.01.2022].

[20] Buffer overflow patching for c and c++ programs: Rule-based approach. *In Proceedings of the ACM SIGAPP Applied Computing Review*, 13:8–19, 6 2013.

[21] Prioritizing alerts from multiple static analysis tools, using classification models. In *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*, pages 13–20. Association for Computing Machinery, 2018.

[22] Cwe - top 25 most dangerous software weaknesses 2021, 2021. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html [Accessed on 03.12.2021].

[23] Mj Muhammad Aslam. Binary instrumentation with qemu. 2016.

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

[25] Fabrice Bellard. Qemu. https://www.qemu.org/ [Accessed on 10.12.2021].

[26] Muhammad Arif Butt, Zarafshan Ajmal, Zafar Iqbal Khan, Muhammad Idrees, and Yasir Javed. An in-depth survey of bypassing buffer overflow mitigation techniques. *In Proceedings of Journal Applied Sciences, 2022*, 12, 2022.

[27] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R Newton. Instruction punning: Lightweight instrumentation for x86-64. *In Proceeding of SIGPLAN, 2017*, 52:320–332, 6 2017.

[28] C Cowan, F Wagle, Calton Pu, S Beattie, and J Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceeding of DARPA Information Survivability Conference and Exposition*, volume 2, pages 119–129 vol.2, 2000.

[29] Jason Deckard. *Buffer Overflow Attacks: Detect, Exploit, Prevent. In Proceeding of the Journal Elsevier Science*, 2005, 2005.

[30] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceeding of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–163. Association for Computing Machinery, 2020.

[31] Jon Erickson. *Hacking: The Art of Exploitation, 2nd Edition*. No Starch Press, 2008.

[32] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. The afl++ fuzzing framework. https://aflplus.plus/ [Accessed on 11.12.2021].

[33] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++ : Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies*. USENIX Association, 8 2020.

[34] Jonathan Ganz and Sean Peisert. Aslr: How robust is the randomness. In *Proceeding of IEEE Cybersecurity Development*, pages 34–41, 2017.

[35] GNU and Fee Software Foundation. Gdb: The gnu project debugger, 1986. https://sourceware.org/gdb/ [Accessed on 10.12.2021].

[36] LLVM Developer Group. The llvm compiler infrastructure project, 2003. https://llvm.org/ [Accessed on 10.12.2021].

[37] Eric Haugh and Matt Bishop. Testing c programs for buffer overflow vulnerabilities, 2003.

[38] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. Fuzzgen: Automatic fuzzer generation. In *Proceedings of the 29th USENIX Security Symposium*, pages 2271–2287. USENIX Association, 8 2020.

[39] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., 1st edition, 2009.

[40] William Klieber, Ruben Martins, Ryan Steele, Matt Churilla, Mike McCall, and David Svoboda. Automated code repair to ensure spatial memory safety. In *Proceeding of IEEE/ACM International Workshop on Automated Program Repair*, pages 23–30. Institute of Electrical and Electronics Engineers Inc., 6 2021.

[41] Chenyang Lyu, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, and Jing Chen. Smart seed generation for efficient fuzzing. *arXiv*, 2018.

[42] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium*, pages 1949–1966. USENIX Association, 8 2019.

[43] Microsoft. Definition of a security vulnerability, 2014. https://docs.microsoft.com/en-us/previous-versions/tn-archive/cc751383(v=technet.10) [Accessed on 18.12.2021].

[44] Linda Null and Julia Lobur. *The Essentials of Computer Organization and Architecture*. Jones and Bartlett Publishers, 2006.

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

[46] James Ransome and Anmol Misra. *Core Software Security: Security at the Source*. CRC Press, 2018.

[47] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection [accessed on 16.04.2022]. 9 2002.

[48] 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.

[49] Fayozbek Rustamov and Joobeom Yun. Deepdiver: Diving into abysmal depth of the binary for hunting deeply hidden software vulnerabilities. *In Proceeding of the Future Internet, 2020*, 12:74, 1 2020.

[50] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. *In Proceeding of the Journal SIGARCH Computer Architecture, 2013*, 41:317–328, 3 2013.

[51] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Tiffany Bao, Ruoyu Wang, Christophe Hauser, and Yan Shoshitaishvili. Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In *Proceedings of the USENIX Security Symposium*, pages 413–430, Boston, MA, August 2022. USENIX Association.

[52] 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.

[53] Michal Zalewski. American fuzzy loop (afl). https://lcamtuf.coredump.cx/afl/ [Accessed on 11.12.2021].