UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Removal of Vulnerabilities in Binary Code by Program Model Checking and Concolic Execution

Luís Pedro Félix Ferreirinha

Mestrado em Informática

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

2024

Acknowledgments

First, I would like to express my gratitude to my advisor, Prof. Ibéria Medeiros, for her belief in my potential. When I was still a Physics student, she saw something in me and welcomed me under her supervision. Her support for my unorthodox ideas, when no one else would, has been invaluable. Without her guidance and encouragement, my achievements in the field of Computer Science would not have been possible.

I also wish to thank all my family, loved ones, and friends for their constant support throughout this journey. A special mention goes to my mother, who especially supported me through tough times, and has been a pillar of strength for me. Her encouragement has been a driving force behind my success.

This work was supported by the European Commission through the H2020 programme under grant agreement 871259 (ADMORPH), and by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020 (https://doi.org/10.54499/UIDB/00408/2020) and ref. UIDP/00408/2020 (https://doi.org/10.54499/UIDP/00408/2020).

For the eternal pursuit of knowledge.

Resumo

Com o avanço da tecnologia nas últimas décadas a dependência que temos desta também aumentou. Desde sistemas de gestão de eletricidade, gás, água e até sistemas de aviação e gestão de tráfego aéreo. Todos estes são geridos por software e o seu funcionamento correto influencia diretamente o bem estar da população. Devido a este avanço, a complexidade dos sistemas utilizados também experienciou um crescimento significativo, o que levou ao aumento da complexidade do software desenvolvido para estes, e consequentemente facilitou o aparecimento de falhas de segurança e vulnerabilidades no código produzido.

A existência destas falhas muitas vezes é ignorada pelos desenvolvedores, que por falta de conhecimento, de tempo ou por ser um produto em fim-de-vida, não apresentam correções para estas. Isto leva a que atacantes motivados procurem a existência destas no software e as tentem explorar para seu benefício. A exploração destas por parte de agentes maliciosos, pode levar a resultados catastróficos, como à perda de dados, destruição de sistemas e possível perda de vida no caso de sistemas críticos.

Estes sistemas são frequentemente construídos utilizando a linguagem de programa C, uma linguagem que permite aos programadores terem acesso à memória de baixo nível, tornando esta uma das linguagem mais populares para sistemas críticos. Devido à sua natureza de baixo nível, esta linguagem também é uma das mais propícias a vulnerabilidades de corrupção de memória, como por exemplo a escrita além dos limites de um *buffer*, que são segmentos de memória com um tamanho pré-definido de elementos. Estas vulnerabilidades geralmente ocorrem quando o programa não assegura que a operação de escrita é efetuada dentro dos limites do *buffer*. A exploração dessas vulnerabilidades pode levar um atacante a obter controlo total do sistema, levando à classificação destas como umas das mais perigosas.

De forma a descobrir estas vulnerabilidades, foram criadas técnicas de análise, entre estas as mais populares pertencem a uma de duas categorias: análise estática e análise dinâmica. A análise estática consiste em analisar o código fonte de uma aplicação, sem o executar; e a análise dinâmica consiste em executar o código da aplicação e analisar o processo de execução e os resultados deste. Ambas estas técnicas têm desvantagens, a análise estática encontra dificuldades com precisão, resultando num número elevado de casos de falsos positivos, em contraste a análise dinâmica, não consegue escalar para aplicações de maior dimensão.

Os problemas encontrados por estas técnicas tradicionais são acrescidos quando estas são aplicadas ao código binário de C. Este é resultado do processo de compilação do código fonte C, um

processo que causa uma grande perda de informação do código original. Devido a este problema, o principal objectivo desta dissertação é criar um novo método, recorrendo a técnicas formais de *model checking* (verificação de modelos) e *concolic execution* (execução concólica), de forma a ultrapassar as limitações das técnicas tradicionais.

O método proposto por esta dissertação consiste em utilizar a técnica de *model checking* para verificar propriedades de segurança na memória pilha de um programa binário, com intento de encontrar vulnerabilidades de *buffer overflow*. Este método foi implementado numa ferramenta denominada BASICS (*Binary Analysis and Stack Integrity Checker System*), que efetua a verificação destas propriedades em ficheiros de programas binários e identifica possíveis vulnerabilidades de buffer overflow procedendo com a correção desta. Para efetuar a correção dos programas a ferramenta recorre a um método de trampolim, que consiste em redirecionar a execução do programa para uma secção de código correta e evitar a secção vulnerável.

De forma a implementar a ferramenta, foi criado um modelo teórico para a memória pilha, este modelo consiste em estados de memória que contêm os frame de pilha ativos para uma dada instância de execução do programa, e por sua vez estes frames contêm o estado de escrita dos bytes daquela pilha. Baseado neste modelo, a ferramenta cria um estado de espaços, iterando por todos os blocos de código do grafo de controlo de fluxo do binário, e fazendo corresponder as instruções de *assembly* a operadores de memória que modificam o estado da pilha. Quando são encontradas chamadas a funções, é efetuada *concolic execution*, de forma a simular o resultado desta chamada na pilha.

Com o espaço de estados gerados, são depois verificadas as propriedades de segurança neste. Estas propriedades são definidas em Lógica Temporal Linear, e modelam o uso correto da pilha de um programa ao longo da execução deste. Para tal foram implementadas propriedades que verificam a integridade do *return address*, do *stack base pointer*, do *stack canary*, e outras que detectam a ocorrência de *buffer underflows*. Qualquer violação destas propriedades, indica a existência de uma possível vulnerabilidade. Sempre que a ferramenta detecta uma violação destas propriedades é emitido um contra-exemplo que contém as instruções que levaram o programa a chegar a um estado inválido. Com base nestes contra-exemplos, é executada uma análise de fluxo invertido de forma a determinar a instrução ou conjunto destas que originaram a vulnerabilidade.

Uma vez detectada a localização das vulnerabilidades, estas são corrigidas, aplicando o método do trampolim. Esta correção vai redirecionar o fluxo de execução do programa para um *patch template*, que são conjuntos de código previamente definidos e compilados, e que foram concebidos com o propósito de limitar o conteúdo escrito para buffers na pilha, impedindo assim as vulnerabilidades de *buffer overflow*, mas mantendo o comportamento original desejado do programa. De forma a garantir que estas correções são válidas, são utilizados *inputs* obtidos durante o processo de *concolic execution*. Estes *inputs* são depois utilizados para testar o programa binário antes e após a correção. Com estes testes, é determinado se a correção aplicada impede o programa de sofrer paragens inesperadas. Se for esse o caso, a correção é considerada como válida.

Para avaliar a ferramenta foi utilizado, um conjunto de programas teste obtidos do NIST SARD

e um conjunto de aplicações de código aberto obtidas dos websites SourceForge, GitHub e Gitlab. Com a análise dos resultados de teste com os programas do NIST SARD, foi possível concluir que a ferramenta é capaz de detectar vulnerabilidades de *buffer overflow* com uma precisão aceitável, que o espaço de estados da memória do programa gerado pela ferramenta refletia com exatidão as operações de memória do código fonte, e que as propriedades de segurança definidas modelam o comportamento de *buffers overflows* destrutivos e permitem a detecção destas vulnerabilidades. Para este conjunto de programas teste, a ferramenta também aplicou correções para as vulnerabilidades detectadas, permitindo concluir que as correções aplicadas são eficazes em remover o comportamento vulnerável dos programas, mas podem comprometer a funcionalidade destes em alguns casos.

Para as aplicações de código livre uma avaliação sobre a performance da ferramenta foi efetuada. Permitindo concluir que esta tem problemas de explosão do espaço de estados, um problema comum em ferramentas de *model checking*, que a impede de efetuar a verificação de binários de maior tamanho.

A ferramenta criada permite aos utilizadores a implementação das suas próprias propriedades se segurança, permitindo assim aumentar as capacidades de deteção desta para além das vulnerabilidades atuais. Possibilitando também modelar comportamentos além de vulnerabilidades, como por exemplo a integridade de certos dados na pilha. Além de ser possível configurar as propriedades também é possível adicionar patch templates. Isto permite a um utilizador expandir a capacidade de correção da ferramenta.

Esta dissertação contribuiu para o avanço das técnicas de descoberta de vulnerabilidades em ficheiros binários com uma nova abordagem baseada em *model checking* e *concolic execution*, e com a criação do BASICS, numa ferramenta customizável de código aberto que implementa esta nova abordagem.

Palavras-chave: Verificação de Modelos, Vulnerabilidades de Overflow na Pilha, Código Binário, Execução Concólica, Análise Estática

Abstract

The C programming language, prevalent in Cyber-Physical systems, is crucial for system control where reliability is critical. However, it is also commonly susceptible to vulnerabilities, particularly buffer overflows, which are ranked among the most dangerous due to their potential for catastrophic consequences. Traditional vulnerability discovery techniques such as static and dynamic analysis, often struggle with scalability and precision when applied directly to the binary code of C. This dissertation introduces a novel approach designed to overcome these limitations by leveraging model checking and concolic execution techniques to verify security properties, defined in Linear Temporal Logic, of a program's stack memory in binary code, and trampoline techniques to fix the identified security issues. The developed tool, BASICS: Binary Analysis and Stack Integrity Checker with Patching, constructs a memory state space from a program's control flow graph and simulates function calls and loop constructs using concolic execution. Security properties defined in LTL model the behavior of buffer overflows, and BASICS identifies these vulnerabilities by analyzing counter-example traces generated when a security property is violated. The tool then addresses these vulnerabilities with a trampoline-based patching method. To ensure the effectiveness of the patches, BASICS tests the patched binaries with crash-inducing inputs extracted during concolic execution, confirming the successful removal of vulnerabilities. BASICS was evaluated using a dataset of small programs from NIST SARD and larger open-source applications. The evaluation demonstrated the tool's effectiveness in detecting and patching buffer overflow vulnerabilities. This dissertation contributes to the field of computer security by introducing a new model checking approach for binary analysis, providing a framework for formal reasoning about stack memory, and delivering a customizable, open-source tool for detecting and patching vulnerabilities.

Keywords: Model Checking, Stack Buffer Overflow, Binary Code, Concolic Execution, Static Analysis

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Chapter 1

Introduction

Software powers the systems of our world, from the smallest gadgets in our homes to the largest machines in our industries. It is essential that this software does not just work, but works without fail, preventing errors that could lead to serious consequences. As our reliance on technology grows, so does the need for software that is not just functional, but secure and dependable.

Most of this software is written in C programming language, particularly in cyber-physical systems where reliability is crucial. C allows programmers to work close to a system's hardware, allowing for greater speed and flexibility, but this comes with significant risks. The language leaves room for vulnerabilities such as buffer overflows, where the lack of safeguards can lead to system compromises and failures. CWE [15] ranks Out-of-bounds Writes as the most dangerous software weakness, this includes Stack-based Buffer Overflow vulnerabilities. These latter ones are especially dangerous, as shown by Aleph One [46], an attacker can hijack the flow of execution of the program and execute arbitrary code, allowing full access to the system.

The task of detecting these vulnerabilities has been a very researched topic, leading to the development of different tools to find these vulnerabilities [33, 30, 51, 37, 4]. These tools employ a variety of different analysis methods, but they often use one of the following approaches: Static Analysis or Dynamic Analysis. Static analysis techniques analyze a program's code without executing it, allowing high code coverage but at the cost of a higher number of false positives [43]. In contrast, Dynamic analysis techniques execute the code of a program [64], this allows these tools to more accurately detect vulnerabilities and achieve a lower rate of false positives but it sacrifices code coverage. Some tools such as Arbiter [61], employ both techniques to achieve higher scalability and precision.

1.1 Motivation

Despite all the security mechanisms and safeguards of modern compilers and operating systems, software vulnerabilities still exist in released C software, i.e., binary programs (or machine-language). When these vulnerabilities are found and reported to the public, the vendors cannot always provide a patch, leading to software that remains vulnerable sometimes upwards of 12 months after the original discovery of a flaw [28].

To mitigate this issue, methods for automatically patching vulnerabilities have been a topic of research. While most methods target source code, there has been progress in binary patching as well. For instance, Ferreira [21] has advanced binary patching through the use of executable instrumentation with a trampoline technique. However, these methods require accurate identification of the vulnerability's exploit vector, a challenging task with disassembled binaries where the program is reduced to Assembly Instructions (the machine-language understandable by humans), offering little insight into the program's higher-level logic.

This leads to the need for a scalable and accurate analysis method to detect vulnerabilities in assembly code. While dynamic analysis offers accuracy, it falls short in scalability for large binaries. Conversely, static analysis scales well but often lacks precision. The question then arises: *can we devise a tool that is both scalable and accurate?*

This dissertation focuses on the development of a static analysis approach capable of detecting buffer overflow vulnerabilities in compiled C binaries and patching them, the focus will be on detecting the vulnerabilities via a formal method called Model Checking and Concolic Execution. Model Checking is a technique for verifying finite state systems, this is accomplished by performing an exhaustive search of the state space of the system to determine if some specification is true or not [39]. Concolic Execution is a hybrid technique that combines Symbolic and Concrete execution that can be used to determine inputs for a program [56].

To detect vulnerabilities using model checking and concolic execution, we will first build a mathematical model representing the program's stack from the disassembled binary, a state space composed of each function's stack frame will then be generated according to a list of transition operators. During this construction, we will simulate C Library function calls and Loops (up to a maximum bound) using concolic execution, to increase the accuracy of the state space. Within this framework, we will specifically model buffer overflow vulnerabilities, along with other potential security issues, to conduct a comprehensive search within the state space. Once a vulnerability is verified, we will implement a corrective patch using the trampoline method detailed in [21].

Model checking is a proven technique for verifying security properties in C programs [12], yet its application to assembly code has been limited due to state space explosion issues. While there have been instances of its use in binary analysis for malware detection [44], showing significant potential, the specific modeling of a binary's stack to verify memory structure properties and detect buffer overflow vulnerabilities remains an unexplored area.

1.2 Objectives

Through this dissertation, we aimed to develop a novel method for detecting buffer overflow vulnerabilities in binary programs and implement this approach in a tool capable of detecting these vulnerabilities, correcting their behavior, and verifying the effectiveness of the applied corrections. To achieve this, we defined several goals to help us reach our final objective.

- Research existing model checking approaches for binary code, focusing on how they model

binary programs and specify their properties.

- Identify how stack buffer overflow vulnerabilities interact with the stack memory of a binary program.
- Develop a model checking approach that allows verification of security properties related to stack memory.
- Research the application of concolic execution to simulate C Library function calls and Loops, in order to enhance the program model.
- Accurately detect stack buffer overflow vulnerabilities by modeling vulnerable behaviors through security properties.
- Automatically patch detected vulnerabilities by rewriting the binary using a trampoline approach.
- Implement the developed approach into a tool and evaluate its performance and effectiveness.

1.3 Contributions

Through the development of this dissertation, we made some relevant contributions to the field of software security, through our research and development of model checking and concolic execution methods for vulnerability discovery in binaries, and patching methods for the removal of buffer overflows in binaries, we will now enumerate and further explain our contributions in detail:

1. **Theoretical Stack Memory Model**: We created a new theoretical model for the stack memory, that allows us to accurately track changes performed on the memory contents throughout the execution of a program. This model introduces new memory transition operators, a representation of program memory states, and a state transition system for the current state of each byte on the stack. By doing so, we established a novel approach to map the execution of assembly instructions directly to operations that modify the stack, providing an accurate reflection of the current state of the stack.
2. **Stack Security Properties**: To detect buffer overflow vulnerabilities, we created new operators for LTL that permit the specification of properties about the stack memory of a program. With these operators, we constructed LTL formulas that model the behavior of typical buffer overflows. With these operators, we also provide a new way for researchers to specify properties about the evolution of the stack memory throughout the execution of a program.
3. **Stack Emulation Approach based on Concolic Execution**: We devised a new approach to emulate the effects of function calls and loops in assembly code on stack memory. Utilizing

concolic execution, our method simulates the execution of these constructs and concretizes the stack memory, allowing us to accurately calculate their impacts on the stack.

4. **Novel Model Checking Approach**: Building on our earlier contributions, we developed a novel model checking approach for binary programs. This approach leverages our stack memory model to construct a state space using memory operators. It then verifies security properties against this state space, employing our LTL operators. This enables the confirmation of the presence or absence of specific behaviors in the stack memory and the detection of buffer overflow vulnerabilities.

5. **Vulnerability Detection and Patch Validation Method**: Utilizing our model checking approach, we developed a novel method for detecting stack buffer overflow vulnerabilities in binary programs by analyzing counter-example traces generated by the model checker. Additionally, we designed a technique for patching these vulnerabilities and validating the patches using inputs derived from the concolic execution process.

6. **Open-Source Tool**: We created a modifiable open-source tool named BASICS: Binary Analysis and Stack Integrity Checker System [1], that implements our previous contributions. This tool is capable of detecting and patching buffer overflows in small binary programs. It allows users to specify custom security properties, custom mappings between properties and CWE classes, and even custom patches, making it highly customizable for individual research needs.

The work conducted for this dissertation led to the publication of our novel approach in the paper titled ”On the Path to Buffer Overflow Discovery by Model Checking the Stack of Binary Programs”, presented at the *19th International Conference on Evaluation of Novel Approaches to Software Engineering* [22].

1.4 Structure of the document

This thesis is organized into seven chapters, each detailing a different aspect of the research:

- Chapter 1: Introduces the reader to the problem, explaining the dangers of buffer overflow vulnerabilities and proposing our solution to address this issue.
- Chapter 2: Defines relevant concepts in this problem area, such as vulnerabilities, model checking, concolic execution, and binary programs.
- Chapter 3: Reviews relevant work previously developed in this field.
- Chapter 4: Presents the BASICS solution in detail and discusses the challenges involved in implementing this approach.

[1] https://github.com/Singularitty/BASICS

- Chapter 5: Details the implementation of BASICS, covering the challenges faced and design choices for each module of the BASICS tool.
- Chapter 6: Presents the results obtained from evaluating BASICS with open-source applications and discusses their implications in detail.
- Chapter 7: Discusses the conclusions drawn from this dissertation, highlighting both positive and negative takeaways, and explores potential future work.

Chapter 2

Background

In this chapter, we will explore the theoretical concepts and technologies behind the research conducted in this dissertation. We will start by discussing software vulnerabilities, with a particular focus on buffer overflows, detailing their causes and potential impacts. Next, we will introduce the concept of Model Checking, providing illustrative examples and an explanation of the formal method. This will be followed by a brief overview of concolic execution. Finally, we will discuss binary programs, specifically introducing the Angr Framework for their analysis and the techniques employed for patching vulnerabilities.

2.1 Software Vulnerabilities

A software vulnerability can be described as a flaw in a program that compromises the program's security and usually of the host system [19]. A vulnerability in a program poses a problem due to the possibility of exploitation by a malicious attacker. Most vulnerabilities start with code that takes data from a third party (attack vector), such as user input or a data packet received over the network.

The result of a bad input can cause a program to stray from the intended execution path, resulting in unexpected behaviors and outcomes. While the most immediate effect might be a program crash, a determined attacker can manipulate the weaknesses to execute arbitrary code, in the form of shellcode, potentially granting the attacker unrestricted access to the host system [46].

These vulnerabilities might exist due to poor implementation logic by the developer, the absence of data validation, or the usage of vulnerable third-party libraries. The severity and exploitation methods of these will vary, resulting in various categories, which include memory corruption, input validation errors, race conditions, and more. A popular method of classification is through the Common Weakness Enumeration (CWE) [1], which provides a comprehensive list of software weaknesses. By categorizing these vulnerabilities, CWE aids developers and security professionals in identifying and mitigating potential security issues.

2.1.1 Buffer Overflow

Buffer Overflows have been identified as the most common and dangerous vulnerabilities to date [15], these occur whenever a program fails to check the bounds of a buffer and allows a write operation to a memory address outside this buffer. These overflows can happen in the heap, a region of memory dedicated to dynamic allocations during runtime, or the stack, a region of memory used to store local variables, function parameters, and function return addresses [46]. Due to the nature of the data stored in the stack, overflows in this memory region, i.e. Stack Overflows, are the most dangerous, since a program relies on the function return address, stored at the top of the stack memory, to have the expected control flow the programmer intended.

Listing 2.1: Stack Overflow Example in C

```
1  void copy(char *str) {
2      char buffer_2[16];
3      strcpy(buffer_2, str);
4  }
5
6  void main() {
7      char buffer_1[256];
8
9      for (int i = 0; i < 255; i++) {
10         buffer_1[i] = 'x';
11     }
12     copy(buffer_1);
13 }
```

The code in Listing 2.1 demonstrates a standard example of a buffer overflow vulnerability. In this code snippet, a 256-byte buffer (`buffer_1`, in line 7) is allocated and filled with the character 'x'. Subsequently, the `copy` function, in line 12, is invoked with `buffer_1` as a parameter. This function initializes a second, smaller buffer (`buffer_2, in line 2` with a size of only 16 bytes and attempts to copy the contents of `buffer_1` into it, by calling the `strcpy` function from the C standard library. The `strcpy` function is considered dangerous, as it does not take into account the size of the destination buffer when copying contents between buffers. Because `buffer_2` is not large enough to accommodate the data from `buffer_1`, this operation results in a buffer overflow, where excess data spills over into the adjacent memory space.

By compiling the previous code to x86-64 Assembly, a low-level machine-language that the Assembler translates into machine code for the process to execute, we gain the ability to analyze the inner workings of the copy function and its interactions with memory.

The assembly code in Listing 2.2 demonstrates the setup of the stack frame of the copy function from Listing 2.1 and why it's vulnerable to a buffer overflow vulnerability. Initially, the stack base pointer (`RBP`) for the previous function is preserved on the stack with the instruction `push rbp`, leaving the register `RBP` free to receive the current value of the stack pointer present in `RSP`, denoted by the instruction `mov rbp, rsp`. Afterward, `RSP` is decreased by 32 bytes with `sub rsp, 32`, allocating space for the local variables of the copy function in its stack frame. Within

Listing 2.2: Copy function's x86-64 Assembly code

```
1  copy:
2          push    rbp
3          mov     rbp, rsp
4          sub     rsp, 32
5          mov     QWORD PTR [rbp-24], rdi
6          mov     rdx, QWORD PTR [rbp-24]
7          lea     rax, [rbp-16]
8          mov     rsi, rdx
9          mov     rdi, rax
10         call    strcpy
11         nop
12         leave
13         ret
```

this space, an 8-byte pointer to `buffer_1` is stored at the address `RBP-24`, and a separate 16-byte space is allocated at `RBP-16` for the contents of `buffer_2`.

When `strcpy` is invoked, it is instructed to copy data from the location pointed to by `RDI` (which currently is `buffer_1`) to the space starting at address `RBP-16` (start of `buffer_2`). Since `buffer_1` contains 256 bytes of data, it far exceeds the 16-byte capacity of `buffer_2`. Consequently, the excess data from `buffer_1` overflows and corrupts the adjacent memory space beyond `buffer_2`. This is depicted in Figure 2.1, where the overflow overwrites other critical data on the stack, such as the previously saved `RBP`. This kind of overflow can lead to the corruption of the stack frame and potentially alow an attacker to take control of the execution flow of the program.

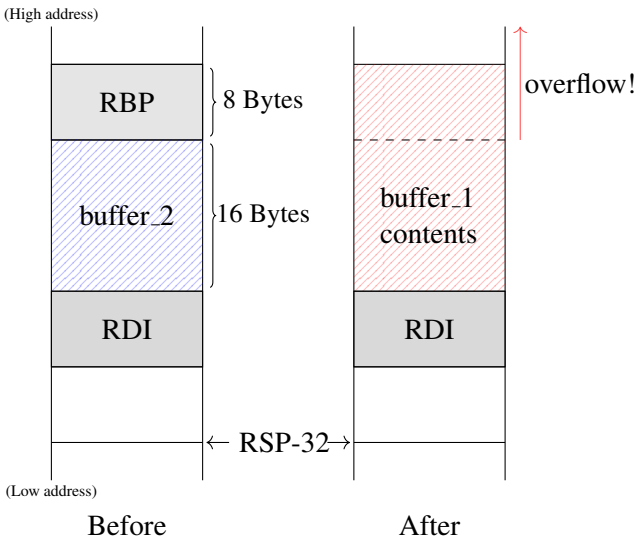

Figure 2.1: Stack buffer overflow of Listing 2.2

2.1.2 The Stack Region

The stack is a dynamic array of memory locations managed at runtime by the kernel, primarily for storing data such as local variables and return addresses. In the x86-64 architecture, the stack pointer register, i.e. `RSP`, indicates the current top of the stack. When data is pushed onto the stack with `push` instruction, the CPU first decrements `RSP` to make room in the stack, then writes the new item at this updated location. Conversely, the `pop` instruction removes an item by reading from the current top of the stack and then incrementing `RSP`. This design means the stack expands downwards to lower memory addresses as items are added, and contracts upwards to higher addresses as items are removed.

The stack is organized into logical sections called stack frames, which are allocated when a function is called and deallocated upon return. Each frame is composed of four key regions [16] (see Figure 2.2)

- **Arguments**: This region holds the current function's arguments, which are pushed onto the stack in reverse order, ensuring the first argument is at the top and easily accessible.
- **Return Address**: The return address is the point to which the caller function will return after execution of the callee function. This address, corresponding to the instruction following the call in the previous function, is pushed onto the stack before transferring control to the callee function.
- **Previous Frame Pointer**: This contains the base pointer (`RBP`) from the caller's stack frame, allowing the called function to reference its caller's frame.
- **Local Variables**: Finally, there is a designated area for the function's local variables. The compiler determines the layout and size of this region based on the program's needs.

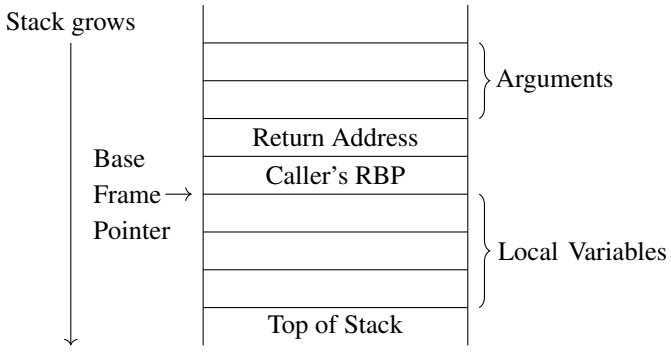

Figure 2.2: Stack Frame layout

The most critical threat posed by a stack overflow is the potential overwrite of the caller's `RBO` and, crucially, the return address. The integrity of the return address is vital, as it dictates where the program's execution should continue after a call. If an overflow allows this address to be overwritten, an attacker can redirect the program's execution flow, possibly to execute arbitrary, and potentially malicious code.

2.2 Model Checking

Model Checking is a computational technique used to analyze the behaviors of dynamic systems, which are represented as state-transition systems [13], This method is extensively utilized in the verification of both hardware and software within the industry. When exhaustive verification of the actual software is infeasible, a simplified model that encapsulates its fundamental design can be created. This model retains the system's essential properties while sidestepping complexities that hinder full-scale verification. Thus, model checking allows for the verification of a system's design when its complete implementation is too complex to verify directly.

According to [13], a model checker is composed of three main components:

- **Model**: A finite state-transition graph that provides adequate formalism for the description of the system, generally designated as a Kripke Structure.
- **Specification**: The system's desired properties are expressed as temporal logic, which provides a framework for specifying the correctness criteria of state transitions.
- **Algorithm**: A computational method used to ascertain if the state-transition model follows the specifications outlined in the temporal logic formulae.

Together, these components facilitate the model checking process. The system is abstracted into a state-transition graph, known as a Kripke Structure, denoted as K. The specifications of the system's behavior are formulated as temporal logic formulas φ. The model checker employs a decision procedure to determine whether $K \models \varphi$ hold, i.e. if the Kripke Structure K satisfies the property φ. Should the K not satisfy φ (expressed as $K \not\models \varphi$), the model checker provides a counter-example, demonstrating how the security property φ is violated within the structure K.

Model Checking Example

To illustrate the practicality and efficacy of model checking, let's consider an example implemented in Promela, a verification modeling language specifically designed to analyze concurrent systems, and used with the SPIN model checker [27] to simulate and verify the correctness of system designs. The code in Listing 2.3 models a simple system with two concurrent processes, A and B, aiming to enter their respective critical sections while avoiding simultaneous execution that could lead to conflicts. Before entering the critical section, each process sets its corresponding flag to true and then checks the other processes's flag to ensure it is not in its critical section. This check acts as a mutual exclusion mechanism to prevent both processes from simultaneously being in their critical sections. After executing the critical section, indicated by a print statement, the process resets its flag to false, allowing the other process to enter its critical section.

A state for this program will be the set of values for the flags together with the location counter which indicates the position in the execution of each process.

$$\text{State} = \{\text{loc}_A = n_1,\ \text{flagA} = \mathit{bool},\ \text{loc}_B = n_2,\ \text{flagB} = \mathit{bool}\}$$

Listing 2.3: Critical Section problem modeled with Promela

```
1  bool flagA = false, flagB = false;
2
3  active proctype A() {
4  do::
5      printf("Non critical section A\n");
6      flagA = true;
7      !flagB;
8      printf("Critical section A\n");
9      flagA = false
10 od
11 }
12
13 active proctype B() {
14 do::
15     printf("Non critical section B\n");
16     flagB = true;
17     !flagA;
18     printf("Critical section B\n");
19     flagB = false
20 od
21 }
```

With a mathematical definition for a program state, we can construct a state transition graph, a graphical representation of the possible execution traces of the program. Each node in this graph corresponds to a reachable state of the program, and the edges represent transitions between these states triggered by the execution of instructions within the processes.

Constructing this state transition graph enables the verification of crucial security properties within our concurrent system, such as mutual exclusion. Mutual exclusion ensures that no two processes access their critical sections simultaneously. To verify this property, we examine if the system can ever reach a state where both processes are in their critical sections, such a conflicting state would be represented as $\{9, F, 19, F\}$, where both processes A and B are in their critical sections simultaneously with both flags set to F. By examining the diagram in Figure 2.3, it is possible to observe that such state is never reachable.

For real-world systems, it is unlikely that the state transition graph can be manually examined. So to check if the system possesses the desired properties, one must use a model checker and specify such properties as temporal logic formulas. For our context, we could use the SPIN model checker to verify if mutual exclusion holds by checking if the following Linear Temporal Logic (LTL) formula holds in every computation:

$$\Box\neg(loc_A = 9 \wedge loc_B = 19)$$

This LTL formula asserts that it is always true that the program counters for processes A and B are never simultaneously at locations 9 and 19, respectively. These locations correspond to the critical sections of both processes. The SPIN model checker can then check this formula against

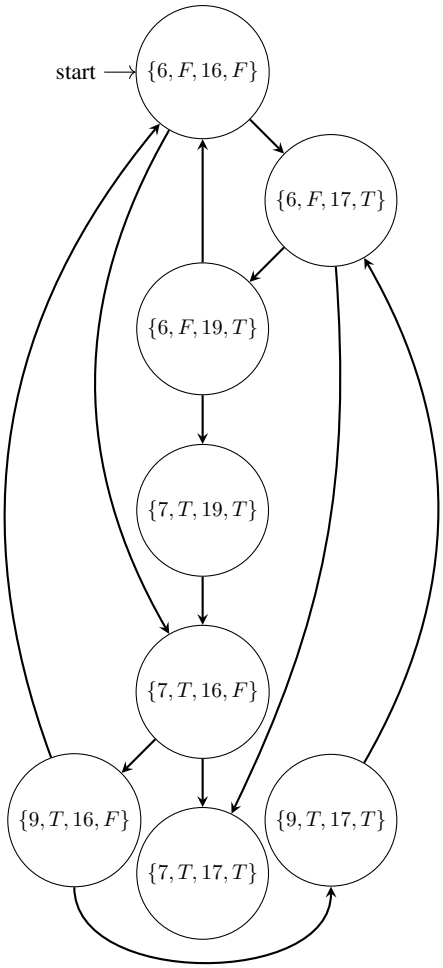

Figure 2.3: State Transition Graph for the system modeled in Listing 2.3

our specified Promela model to verify if mutual exclusion holds in every possible computation, returning a counter-example trace if it does not hold.

2.2.1 Linear Temporal Logic

Temporal logic is used to reason about the way the world changes over time. In the context of software, it is used in the specification and descriptions of systems by describing the evolution of states of a program which gives rise to descriptions of executions. There are different types of temporal logics, depending on their perspective on the progression of time, the two main ones are linear and branching [13].

In the linear perspective, time is seen as a sequence of distinct moments, with each system execution represented as a series of states. When a system has multiple possible execution paths, it is viewed as separate possible execution traces and the system has a set of possible behaviors.

Proposition Linear Temporal Logic (LTL) as the name implies, follows the linear-time view. In addition to the operators present in proposition logic, this logic provides temporal operators that connect different stages of the computations and talk about dependencies and relations between

them. LTL formulas are constructed using normal Boolean operators ($\neg$, $\vee$, $\wedge$) and the temporal operators *next*, *previous*, *until* and *since* [13].

- $\bigcirc\varphi$ (Next): Refers to the immediate subsequent state in a computation.
- $\ominus\varphi$ (Previous): Points to the immediately preceding state.
- $\varphi_1 \mathcal{U} \varphi_2$ (Until): Describes a condition that must hold until another condition becomes true.
- $\varphi_1 \mathcal{S} \varphi_2$ (Since): Indicates that its first operand holds at all points in the past until some past point where its second operand holds

These operators can then be used to define temporal abbreviations which are the most commonly used operators in LTL formulae:

- $\Diamond\varphi$ (eventually): This operator is used to specify that a certain condition is expected to be true at some point in the future. It asserts that there exists a future state in the execution where the condition holds.
- $\Box\psi$ (always): This operator signifies that a condition must hold in all states of execution. It is used to express invariance.
- $\varphi \mathcal{W} \psi$ (Weak-Until): Asserts that a certain condition must hold up until another condition becomes true. However, unlike $\mathcal{U}$, it does not require that the second condition ever becomes true, allowing the first condition to remain true indefinitely.
- $\varphi \mathcal{R} \psi$ (Release): Similarly to $\mathcal{U}$, this operator asserts that a condition must hold until, and including to the point where the second condition becomes true. However, if the second condition is never found to be true then the first must hold forever.

2.2.2 ω-automaton

Before defining what a ω-automaton is, we must first define what an automaton is. These can be described as a model of a machine that responds to a predetermined sequence of inputs. There are several types of automaton, the simplest being a finite state automaton [49] which can be defined as follows [27].

Definition 2.1 (Finite State Automaton)**.** *A finite state automaton (FSA) is a tuple* $(S,\ S_0,\ \mathcal{L},\ \mathcal{T},\ \mathcal{F})$ *that consists of the following components:*

- S *is a finite set of states.*
- S_0 *is a initial state, such that* $S_0 \in S$*.*
- $\mathcal{L}$ *is a finite set of labels.*
- $\mathcal{T}$ *is a set of transition, such that* $\mathcal{T} \subseteq (S \times \mathcal{L} \times S)$*.*

- *$\mathcal{F}$ is a set of final states, such that $\mathcal{F} \subseteq S$.*

A run for an FSA will then be an ordered, and possibly infinite, set of transitions, e.g., $\{(s_0, l_0, s_1), (s_1, l_1, s_2), ...\}$. These runs can further be considered as accepting runs, if it terminates in a final state of the automaton, i.e., the final transition of the run leads to a final state.

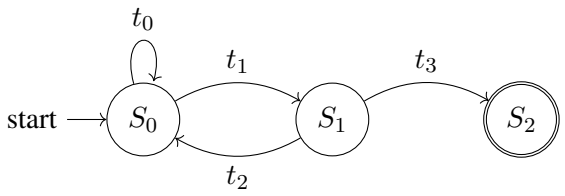

Figure 2.4: Finite State Automaton

An example of an FSA is illustrated in Figure 2.4, in this automaton, S_0 is the initial state and S_2 is the final state. A valid accepting run would be, for example, $\{(S_0, t_1, S_1),\ (S_1, t_3, S_2)\}$.

An ω-automaton is a variation of an FSA that takes infinite strings as input, and instead of having a set of accepting states, it has a variety of acceptance conditions. An infinite run in this type of automaton is called ω-run, for these to be accepting runs there must exist some final state that is visited infinitely often in the run. Any finite run can be extended to an infinite ω-run, through a stuttering rule. This rule consists of adding a no-op operation that is always executable and has no effect.

There are several classes of ω-automata with different accepting conditions, including Büchi, Rabin, Streett, parity, and Muller automata, each of these can be deterministic or non-deterministic.

These types of automata can be particularly useful for Linear Temporal Logic Model Checking since LTL formulas can be translated to ω-automata. This translation allows the Model Checking problem to be formalized as follows.

1. Translate the LTL formula into a ω-automaton B;
2. Compute the interleaving product A of the FSA representing the space state;
3. Compute the synchronous product P of $A \times B$;
4. Search for accepting runs of the automaton P, using the corresponding ω-automaton accepting rule.

The SPIN model checker utilizes the algorithms described in [24, 20] to convert LTL formulas to Büchi automata. An example of this process can be explored by translating the LTL formula $\Diamond\Box p$ to a Büchi automaton, this can be performed utilizing any of the algorithms in [24, 20, 23]. The result is the automaton in Figure 2.5

2.2.3 Satisfiability Modulo Theories

The Boolean Satisfiability Problem (SAT) is a problem in computer science that involves determining whether a given propositional formula, such as $(x \vee y)$, can be satisfied. This means finding

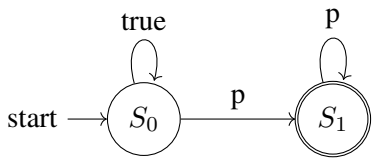

Figure 2.5: Büchi Automaton for $\Diamond\Box p$

a variable assignment that makes the formula evaluate to true. Programs designed to solve these problems are known as SAT solvers, and they output whether a formula is satisfiable or not.

Satisfiability Modulo theories (SMT) are a generalization of boolean satisfiability (SAT) to more complex formulas by adding equality reasoning, arithmetic, and other first-order theories. These problems can be viewed as the problem of determining whether a mathematical formula is satisfiable or not, and the programs that solve these are called SMT Solvers.

These solvers are integral to some formal techniques in computer science, such as Symbolic Execution and Bounded Model Checking. These usually involve gathering constraints in the form of Boolean formulas or mathematical formulas, which are then solved using an SMT Solver. One popular SMT Solver is z3 [14], which was developed by Microsoft and is incorporated into many tools such as the verification-aware Dafny programming language [35] and the binary analysis framework Angr [59].

2.3 Concolic Execution

Concolic Execution is a method that combines both symbolic and concrete execution, meaning that symbolic values and concrete values are used for inputs and the program is both executed concretely and symbolically. The concrete execution part of concolic execution constitutes the normal execution of the program, while the symbolic execution one collects symbolic constraints over the symbolic input values at each branch point encountered along the concrete execution path [55].

The process starts with executing the program on a set of initial inputs. As the program runs, it collects constraints on the inputs from conditional statements encountered along the execution path. These constraints are then used to generate a symbolic representation of the program's execution, capturing the relationships between inputs and the program's behavior. To solve these constraints and determine if a path is executable SMT Solvers are used.

2.4 Binary Programs

For a typical program written in the C programming language to be executed, it must first be translated into a format that the CPU can execute. This translation is done through a process known as compilation. Compiling a C program involves several steps: preprocessing, compiling, assembling, and linking. It is a complex procedure that results in an executable binary file where most of the information from the source code is stripped away. This stripping of information is

one of the factors that makes vulnerability discovery in binary programs challenging.

Depending on the target operating system, the format of the binary will change, with the two most common formats being Executable and Linkable Format (ELF) for Linux and Portable Executable (PE) for Windows [2].

2.4.1 Angr Framework

Angr [59] is an open-source binary analysis platform for Python, it provides an extensive toolkit to address a wide range of binary analysis tasks, such as symbolic execution, concolic analysis, and binary instrumentation for a variety of different architectures, including x86, ARM, and MIPS, among others. Due to its development being primarily in Python, it offers a flexible and approachable platform for analyzing binaries.

The initial step in Angr's analysis process involves disassembling the binary. Disassembly, the task of translating binary code back into assembly instructions, is inherently challenging due to the loss of some high-level information during the original assembly process. Angr tackles this challenge by employing a recursive disassembly strategy, striving for the most accurate reconstruction of the assembly code. Post-disassembly, Angr leverages the Capstone disassembly framework to provide a detailed programming interface to each assembly instruction, facilitating deeper analysis [47].

One of Angr's most powerful features is the Simulation Manager, this tool allows one to control and manage the exploration of execution paths in a binary. In a Simulation Manager, states are organized into stashes, which can be stepped forward, filtered, merged, or moved around as the user wishes. Through the use of stashes, custom exploration techniques can be used to categorize and find specified states (e.g. a state that reaches a certain address), while pruning all states that do not meet the desired requirements.

Additionally, Angr includes a built-in analysis functionality to construct a control flow graph (CFG) of a binary program. The CFG is a graphical representation where nodes are basic blocks, and contiguous sets of assembly instructions, and edges represent control flow transitions like jumps, calls, and returns.

2.4.2 Binary Patching

Binary patching is the process of altering a binary file with the intent of removing critical security issues. As software grows in complexity and more vulnerabilities are uncovered, the need for effective binary patching has become increasingly pronounced. These vulnerabilities may remain unaddressed for extended periods, as the vendor cannot provide a fix, or the software in question is legacy and no longer supported [28]. In these cases, binary patching emerges as a crucial intervention method. It allows for direct modifications to the executable code of the software, bypassing the need for source code access. This approach is especially vital when conventional patching methods are not feasible, making it an indispensable tool for maintaining software security. By applying binary patches, vulnerabilities can be remediated, thus safeguarding the software against

potential exploits and breaches.

One effective approach to binary patching involves using a trampoline mechanism to redirect a program's control flow, a method employed by Diogo Ferreira [21] to address buffer overflow vulnerabilities in binary files. To achieve this, templates are created for supported vulnerable function calls, these templates are small pre-compiled C applications. These templates are then added to the binary through the replacement of the vulnerable call by an e9 jump instruction, this instruction redirects the control flow of the program to a trampoline which consists of the patch followed by a jump instruction that redirects the control flow once again back to the main application. This is exemplified in Figure 2.6.

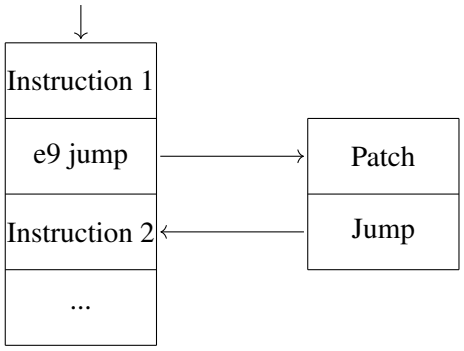

Figure 2.6: Simplified example of the trampoline method. Program control flow follows the arrows

Chapter 3

Related Work

In this chapter, we will provide an overview of the current works and techniques in the research areas of vulnerability discovery, model checking in software security, and code repair. We will specifically focus on applications in both source code and binary code contexts for each research area. This review will highlight the state-of-the-art approaches, and discuss their strengths and limitations

3.1 Vulnerability Discovery Techniques

Detecting software vulnerabilities is a long-standing and well-researched problem in the area of security. Most of this research is targeted at discovering vulnerabilities in source code, most commonly in mainstream languages such as C/C++, PHP, Java, and Python. Only a smaller, although still significant, portion of this research is aimed at binary programs.

This body of research utilizes a plethora of methods and techniques, such as static and dynamic analysis, and machine learning. Several survey studies have been conducted on the published state-of-the-art techniques and popular available tools, such as [32], where the authors performed a comparative study of static analysis tools for C/C++ and Java code, highlighting classes of vulnerabilities that remained undetected by these tools. For machine learning approaches, [58] summarized the current research landscape of the techniques focused on source code, identifying the most commonly used datasets to train the ML models.

3.1.1 Detecting Vulnerabilities in Source Code

For the C language, CorCA [30], combined static and dynamic analysis to detect buffer overflows in C source code, the detection method involves identifying code slices with potential vulnerabilities, compiling them, and performing fuzzing. For PHP, [3] identifies vulnerabilities by performing graph traversals on code property graphs of PHP applications. Recent research has increasingly focused on utilizing machine learning techniques to detect vulnerabilities. Most commonly, these works [5, 51, 25], either train models to detect vulnerabilities, directly on the lexed sourced code or create embeddings from the source code. One particular work [40], utilizes natural language processing to discover and identify vulnerabilities in PHP source code.

3.1.2 Vulnerability Discovery in Binary Programs

The detection of vulnerabilities in binary code is a far more challenging problem, due to the loss of information that occurs during the compilation process of source code to machine language. Despite this, there are been some significant contributions to this field, Arbiter [61], combined both static and dynamic analysis in its approach, allowing for the detection of multiple classes of vulnerabilities. Vyper [6], another example capable of multiclass vulnerability detecting, leverages concolic execution and analyzes sensitive memory zones. To detect Integer Overflows, IntScope [63] converts the disassembled code to an IR (Intermediate Representation) and performs taint analysis and symbolic execution, and [67] utilizes pattern matching, and dynamic symbolic execution (DSE). Additionally, machine learning techniques have also been leveraged to discover vulnerabilities in binaries. For example. VulkHawk [38] and [52], created embeddings of the disassembled code, and trained language processing models with these embeddings to detect multiple classes of vulnerabilities.

3.1.3 Fuzzing

Fuzzing is one of the most popular methods to discover vulnerabilities in software, it consists of generating test cases for an application, generally abnormal stochastic inputs, with the intent to cause a program crash and detect potential bugs. The study [36], compiled recent advances in fuzzing solutions, going in-depth at covering coverage-based fuzzing techniques, and discussing the usage of static analysis, symbolic execution, and machine learning to improve the fuzzing process. One particular work of interest [55], presented a novel test case generation method, called grey-box concolic testing. This method leverages lightweight instrumentation to generate high-coverage test cases high-coverage test cases for binary programs. The authors evaluated their solution against state-of-the-art fuzzers, such as AFLFAST [8] and LAF-intel [34], finding that it excels in terms of both code coverage and bug finding.

3.2 Model Checking in Software Security

Model Checking is a formal method traditionally used to model and study software and hardware behavior, generally focusing on verifying the existence of certain functionalities or the absence of unwanted behavior such as deadlocks. There are several model checking approaches, the two most common being explicit state-based and constraint-based model checking, commonly known as Bounded Model Checking (BMC).

The direct application of this formal method for discovering vulnerabilities is not often found in literature, but some works delving into this application exist. For web security, [29] utilized BMC to verify the source code of web applications, and in systems security, [54] model checked an entire Linux distribution to find exploitable bugs.

3.2.1 Formal Verification of C programs

Some works in literature present tools to verify C source code, some of the most notable will are highlighted here. MOPS [10], was presented as a tool to verify security properties in C software. The authors model the target program for verification as a pushdown automaton and represent security properties as finite state automata. These are then verified against the model of the program by utilizing explicit state model checking techniques, to determine the reachability of risky states in the pushdown automaton. By utilizing this formal method, MOPS is not only capable of determining the presence of vulnerabilities but also verifying their absence in the program. The tool was later utilized in a different work [9], to model check UNIX applications and discover security flaws, model checking over a million lines of C code in the process.

CBMC [12], was presented as a tool that utilized BMC to formally verify ANSI-C programs. It allows for the verification of memory safety, which includes array bounds checks and safe usage of pointers, and it also allows verification of exceptions and user-defined assertions. The authors reduced the Model Checking problem to determining the validity of a bit vector equation, this is accomplished by closely unwinding all loop constructs and backward goto statements, and expanding function calls, transforming the program into a static single assignment (SSA) form. The result produces two bit-vector equations, one for the constraints (C) and another one for the properties (P). To verify a property, the equation ($C \wedge \neg P$) is converted into Conjunctive Normal Form (CNF) and passed to a SAT solver, and if the equation is satisfiable, the property is found to be violated, otherwise, it is found to hold.

3.2.2 Model Checking Binaries

Although not commonly used in binary code due to the state explosion problem [62], model checking has been used to detect malware behaviors, and validate micro-controller code [41, 50, 53].

To detect malicious behaviors in binaries, [44], proposed SPCARET a new temporal logic, to model malicious behaviors and an efficient algorithm to model check SPCARET formulas against Pushdown Systems. To model binary programs, the authors utilize Pushdown Systems adapting them to their needs by keeping track of call and return actions in each path, naming this model Labelled Pushdown Systems. Malicious behaviors are then specified as SPCARET formulas, an extension of the linear temporal logic of CAlls and RETurns (CARET), these behaviors include, Opening and listening to a specific port, and Registry Key Injection.

More related to vulnerability discovery, the framework HeapHopper[18], analyzes the exploitability of different heap implementations, by leveraging BMC and symbolic execution. HeapHopper works by finding sequences of transitions performed in the program that make the constructed model of the heap implementation reach states that invalidate specific security properties. The researchers defined the transitions as operations that can modify the heap, either directly or indirectly, considering the following as possible heap operations: malloc, free, overflow, use-after-free, double-free, and fake-free. Sequences of transitions are then created and checked against the heap model, by using the Angr framework as a symbolic execution engine. If a violation is found,

HeapHopper outputs a proof-of-concept code that can be used to study the security violation.

3.3 Code Repair

Detecting vulnerabilities is one of the most researched topics in software security, leading to the report of many zero-day vulnerabilities in software. To address these developers must be able and willing to fix the reported vulnerabilities, which is often a time-consuming task. Addressing this, automated code repair aims, to fix vulnerabilities without human intervention [66].

To categorize existing repair techniques, Monperrus [42] contributed a comprehensive survey categorizing these into two principal classes: behavior-based and state-based. Behavior-based techniques primarily focus on modifying the source or binary code to alter a program's operational behavior. In contrast, state-based approaches involve changing the program's state during runtime, such as altering the input, stack memory, or heap memory. Complementing this categorization, Pinconshi et al. [48] compared several state-of-the-art techniques, assessing their effectiveness. The author's findings suggest a higher efficacy in techniques that either employ brute-force search strategies or execute functionality deletion in a brute-force manner.

3.3.1 Source Code Repair

Tackling the repair of C source code, CorCA [30] identifies and extracts program slices classified as vulnerable, and substitutes C-library function calls susceptible to vulnerabilities with safer counterparts. Additionally, it refines this process by appropriately adjusting the arguments passed to these functions.

Recently other works have developed code language models for code repair, Jiang et al. [31] evaluated these models, and showed that fine-tuned CLMs significantly outperform traditional techniques in bug fixing and efficiency.

3.3.2 Binary Patching

For binary programs, the problem of automatic repair is more challenging than in source code, with most tools and methods relying on heuristic-based recovery of control flow information from binaries, which tend to scale poorly. Addressing this issue, E9Patch [17] was presented as a tool adept at statically rewriting x86-64 binaries. It utilizes a range of control flow-agnostic rewriting techniques, including instructing punning, padding, and eviction. A feature of this tool is the ability to insert jumps to trampolines without the need to relocate other instructions within the binary, thereby improving its ability to handle large binary files efficiently.

Chapter 4

BASICS Solution

In this chapter, we introduce our proposed solution, BASICS: Binary Analysis and Stack Integrity Checker System. We will begin by outlining the key challenges we encountered during the design of BASICS. Following this, we will present the overall architecture of BASICS, providing a comprehensive overview of its components. Finally, we will go into the details of each module that constitutes BASICS, explaining their roles, functionalities, and how they work together to achieve the desired outcomes.

4.1 Challenges

Developing a scalable and precise tool using formal methods to detect buffer overflow vulnerabilities in x86-64 binaries presents significant challenges. The transition from theory to practical implementation is complete with numerous potential pitfalls and obstacles. To provide context for our proposed solution, we will first discuss the challenges involved.

4.1.1 Performing Model Checking on x86-64 binaries

Applying model checking techniques to binaries presents significant challenges, primarily due to the state explosion problem. The vast amount of information required to track and the large number of branches present in compiled code can cause the state space to grow exponentially, and consume excessive memory resources. This large state space also leads to a potentially lengthy model checking process.

To mitigate this issue, we designed a simple model that will serve as the basis for constructing the state space, keeping the number of transition operators small, to avoid creating too many states, and using a memory-efficient implementation.

Another critical issue is the problem of disassembly. Current disassembly techniques are imperfect, meaning that the disassembled code may not accurately represent the assembly code generated by the compiler. This issue is generally less severe for smaller to medium-sized binaries. However, for larger binaries that have been stripped to prevent reverse engineering, disassembly becomes extremely difficult, if not impossible.

Due to these considerations, we opted for a disassembling solution that utilizes a recursive disassembly algorithm rather than a linear one. This algorithm significantly improves disassembly accuracy and handles obfuscated binaries more effectively. However, we decided not to support stripped binaries, as they lack the necessary information for our approach to function properly.

4.1.2 Detecting Vulnerabilities through Model Checking

Detecting vulnerabilities through model checking requires careful consideration of two key aspects. First, the model of the system must be sufficiently detailed to represent potential malicious traces. Second, the properties verified against this model must ensure that there are no possible traces leading to these malicious states, meaning that when one such trace is detected, a potential vulnerability might be present in the program.

The challenges associated with these considerations include determining the appropriate level of detail for the model. If the model is not accurate enough, there is a risk of failing to detect vulnerabilities. Conversely, if the model is too detailed, it may lead to a state explosion problem. For the security properties, the challenge lies in modeling the vulnerability behaviors using temporal logic, which may involve creating custom operators and functions to accurately detect these vulnerabilities. To address this challenge, we constructed a memory model from the ground up to accurately represent the state of bytes in the stack memory of a binary. We developed operators for LTL to address individual bytes. This approach provides a balance between the accuracy and complexity of the model.

4.1.3 Performing Concolic Execution on Binary Code

One significant technical challenge addressed in this dissertation was the application of concolic execution techniques to binary code, particularly to emulate the effects of function calls. Unlike source code, binary code lacks the high-level information and abstractions that facilitate straightforward symbolic analysis. To extract detailed information regarding changes to the stack, we needed a method to accurately calculate the modifications made by function calls.

To achieve this, we explored existing solutions and selected Angr's symbolic execution engine. This choice allowed us to set hooks on states containing function calls and emulate these calls, thereby obtaining a concrete representation of the stack's state after emulation.

4.1.4 Patching Vulnerabilities in Binary Programs

The final challenge we faced was the removal of detected vulnerabilities in binary programs. Unlike source code, we cannot simply rewrite the binary with corrected assembly instructions. Instead, we must consider aspects such as instruction addresses, library linking, and other low-level details to successfully modify the binary. There are several existing tools available for this task, each with different requirements and approaches.

After a thorough evaluation of these solutions, we selected E9Path [17], which met our specific needs. E9Path facilitates effective binary patching through the use of the trampoline approach.

This method allows us to create patch templates that can address vulnerabilities on an individual basis, ensuring that the patches are applied correctly and consistently across different binaries.

4.2 BASICS a Model Checker for x86-64 Binaries

BASICS is the solution proposed by this dissertation. It aims to find and repair stack buffer overflow vulnerabilities in binary programs. To find the vulnerabilities, we propose the usage of model checking to verify security properties of the program stack memory, these properties model the correct usage of the stack memory space, and a violation of these would account for a potential vulnerability, besides detecting vulnerabilities the model checker would also allow the verification of user-defined security properties via Linear Temporal Logic formulas, allowing to verify any other desired property of the stack memory.

To verify the specified security properties, we construct a theoretical model of the stack memory to generate a state space of the program's stack. This state space is initially generated based on memory write operations, identified through defined transition operators. The state space is then further refined using concolic execution to emulate function calls and loops, which further increases its accuracy. Upon completion of this construction, the model checker conducts a comprehensive search within the state space to identify any traces that violate the specified properties.

At the end of the model checking process, a report is emitted, if all security properties are found to hold a document with the verified properties is emitted. If at least one security property is found to be violated, documents with the violated properties, respective counter-example traces, and concolic inputs are emitted. For the latter case, the binary then undergoes a repair phase if the violated properties are found to correspond to a fixable known vulnerability.

To repair any detected stack buffer overflow vulnerabilities we propose the usage of a trampoline to patch the binary, redirecting the control flow of the binary program through a jump instruction to a patch template containing the correct code and avoiding the vulnerable instructions.

After repairing the detected vulnerabilities, we propose verifying the correctness of the patched binary by testing it with inputs extracted from the concolic execution process. These inputs, which lead to the violation of security properties, are considered malicious. By testing both the original and the patched binaries, we can determine the effectiveness of the patch in fixing vulnerabilities that cause program crashes.

Figure 4.1 provides an overview of BASICS's architecture, which is composed of the following integral modules:

1. Binary Data Extractor
2. Model Checker
3. Security Property Converter
4. Vulnerability Patcher and Validator

Detailed descriptions of each module are provided in the subsequent subsections.

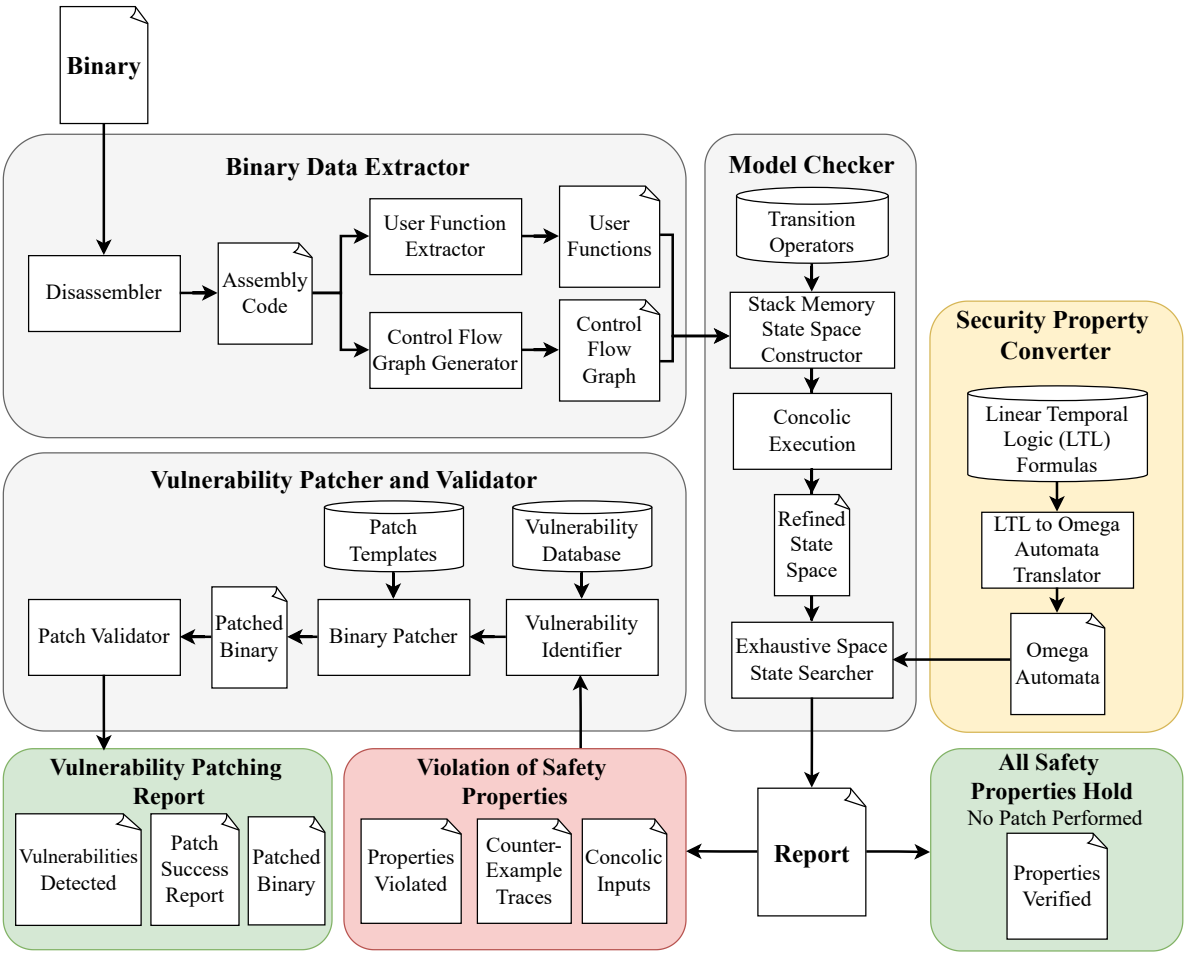

Figure 4.1: Architecture for BASICS

4.2.1 Binary Data Extractor

The Binary Data Extractor module begins by taking a binary program and disassembling it through a recursive disassembly process. This approach is chosen for its enhanced accuracy in extracting x86-64 assembly code. Once the code is obtained, the module performs two critical analyses:

- **Control Flow Graph Extraction:** This analysis involves the extraction of a control flow graph from the program. The information obtained from this process is essential for constructing the program's state space, which is crucial for subsequent modules in the architecture.

- **Function Identification:** This second analysis is focused on pinpointing all user-defined functions within the code. It extracts vital details such as the names and addresses of these functions.

4.2.2 Security Property Converter

The Security Property Converter module functions as an interface within our architecture, facilitating the specification of additional security properties by users. These properties are crucial for the verification of a given binary, especially for customized security needs. Here is a description of how the module operates:

- **LTL Formula Database:** This database stores the user-specified security properties formulated as Linear Temporal Logic (LTL) formulas. It also includes predefined properties that specifically model stack buffer overflow vulnerabilities, and are crucial for detecting these types of vulnerabilities.
- **Omega Automata Constructor:** This component is essential for translating the LTL formulas into Omega Automata. The translation is a prerequisite step before the Model Checker can verify the properties against a given memory state space.
- **Omega Automata:** After conversion, the LTL formulas, now in the form of Omega Automata, are stored here. These automata are then forwarded to the Model Checker module, where they undergo a verification process to determine if they hold for the analyzed program.

4.2.3 Model Checker

This component plays a crucial step in the proposed solution, it is in charge of creating the state space for the program, performing the model checking process, and verifying security properties of the program stack memory, to do this it operates in several key stages:

- **Stack Memory State Space Constructor:** Initially, the module constructs a state space model. This is achieved by utilizing a database of transition operators, which defines which assembly instructions can change the state of the stack memory within the program. The module iterates through the basic blocks of the previously generated CFG and matches each assembly instruction in the block to the defined memory operators.
- **Concolic Execution:** During the creation of the state space, concolic execution is used to emulate C standard library function calls and loops, simulating their effects on the program stack memory. These changes are then reflected in the stack memory state. This technique combines concrete and symbolic execution to obtain symbolic constraints on the stack and concrete values when possible. This approach increases the precision of the state space and ensures a more accurate representation of the program's behavior.
- **Exhaustive State Space Searcher:** The final stage involves verifying security properties, which are represented by omega automata. This module conducts an exhaustive search of the product of the state space and the omega automaton to determine whether each branch of the program's possible execution traces ends in an accepting state of the omega automaton.

If a branch ends in a non-accepting state, that trace violates the property and is emitted as a counter-example trace for that specific property. This process is essential to ensure that the program aligns with predefined security standards and is free from certain classes of vulnerabilities.

Following the completion of the model checking process, the Model Checker module generates a detailed report. The contents of this report vary depending on the outcomes of the verification process and the status of the binary program. The scenarios and corresponding outputs are as follows:

1. **Violation of Security Properties:** If any security property is found to be violated, the report includes:
   - A document listing the violated properties
   - A document detailing counter-example traces, which depict the sequence of operations leading to an invalid state in the stack.
   - A document containing the concretized inputs obtained during the simulation of `stdin` functions from the C standard library.
2. **All Properties Verified:** If all security properties are verified, the report contains:
   - A document enumerating the verified properties.

4.2.4 Vulnerability Patcher and Validator

When at least one security property is found to be violated in the program's memory state space, the binary is automatically forwarded to the Vulnerability Patcher and Validator module for further processing. This module is divided into three primary components:

- **Vulnerability Identifier:** The first step in addressing vulnerabilities is pinpointing their exact source within the binary code. This process involves a two-phase approach. Initially, the type of vulnerability is determined by correlating the violated security properties with entries in a vulnerability database. Subsequently, a reverse-flow analysis of the counter-example traces is conducted to locate the precise position of the vulnerable C standard library call in the program's code that led to the vulnerability.
- **Binary Patcher:** Once the type and location of the vulnerability are identified, this component modifies a copy of the original binary. It employs a trampoline mechanism, which redirects the program's control flow to a patch template designed to circumvent the identified faulty behavior. To ensure effective patching, a variety of patch templates, each tailored to specific unsafe C standard library functions, are maintained in a database. Users can expand the range of functions that can be patched by adding additional patches to this database, which the patcher will automatically utilize.

- **Patch Validator:** Once the patching process is complete, the resulting binary is tested alongside the original binary to verify if any buffer overflow-induced crashes were corrected. To test the binaries, we use inputs generated during the state space creation by the concolic execution process. These inputs, which can lead to invalid states and expose vulnerabilities, are considered malicious. If the original binary crashes with these inputs, but the patched version does not, the patch is considered successful.

Once the patching process is complete, the revised binary is returned along with a patch success report. This report contains the inputs used to test the binaries, the results of executing those binaries, and a list of potential vulnerabilities detected in the original binary.

Chapter 5

BASICS Design and Implementation

In this chapter, we will discuss the design and implementation of BASICS. We will begin by exploring some of the design choices made for the approach, followed by a detailed explanation of how each module works and was implemented. Additionally, we will address the specific challenges encountered during the implementation of each module.

5.1 Framework Choices

Before starting the implementation of the proposed solution, several key decisions had to be made. Primarily, we needed to choose an appropriate disassembling platform and decide whether to use an existing model checker or to build our model checking solution.

5.1.1 Angr Framework

Starting with the most important choice, the backbone of our implementation: we decided to utilize the Angr Framework [59] as our disassembling platform. Angr is a binary analysis framework for the Python programming language that uses the Capstone disassembler as a backend to disassemble binaries.

We chose this platform due to its modularity and ease of integration with Python code, as well as its built-in powerful features, such as concolic execution. By selecting a framework that encompassed all the necessary features for our approach, we avoided the complexity of integrating multiple solutions, potentially built for different programming languages. Another factor influencing our decision was Angr's proven track record. It has been used in many state-of-the-art binary analysis tools, such as Arbiter [61] and HeapHopper [18].

5.1.2 Why not use SPIN?

The second major decision we faced was whether to build our own model checker or use an existing one. Since none of the currently available model checkers met our needs, we would have had to significantly adapt our approach to integrate another model checker.

One model checker we considered was SPIN [27], one of the most powerful and popular options available. SPIN is highly efficient and fast, incorporating decades of progress in the field.

However, to use this model checker, we would have needed to build our state space and convert it to the Promela language. This task would be time-consuming and likely not result in a one-to-one translation, as Promela is designed to model concurrent processes, not the stack memory of binary programs.

Therefore, we decided to build our own model checker. We implemented the algorithms described in [13] and utilized a freely available tool to convert our LTL formulas into omega automata. The downside of building our own model checker was performance, as our solution could never match the efficiency of SPIN. However, we anticipated that the concolic execution process would be the primary performance bottleneck in our solution, so this was not a major concern.

5.2 Extracting data from the binary

The first step in our approach is to take an input binary file, disassemble it using Angr, and extract a CFG and the user functions from it.

To perform this analysis, we create an Angr project and point it to the location of our binary file. We specify that we wish to automatically load the simulated procedures of the C library, but exclude certain C library functions from being loaded for symbolic execution later on during the space state generation. The full list of unloaded functions can be found in listing 5.1. We found these functions, did not contribute to the creation of the stack memory state space, only slowed down the concolic execution process, and created state explosion issues, particularly the functions related to handling files, since these required a large amount of memory to deal with symbolic file pointers and symbolic files.

Listing 5.1: Unloaded C-Library Functions

```
"free", "printf", "puts", "strlen", "printf", "fprintf", "fopen",
↪  "fclose", "fscanf", "strcmp", "system", "exit", "time", "error",
↪  "perror", "fwrite", "printf_unlocked", "puts_unlocked",
↪  "putchar_unlocked", "fputs_unlocked", "fputc_unlocked",
↪  "fprintf_unlocked", "stack_chk_fail"
```

5.2.1 Obtaining the program's CFG

After loading the binary with Angr and creating a project, we can use one of the built-in analyses to easily extract a CFG of the binary program. The framework offers two CFG recovery analyses: *CFGFast* and *CFGEmulated*.

CFGFast utilizes static analysis to generate the CFG. While it is the faster option, it has accuracy issues because some branches can only be resolved during execution. The second analysis, *CFGEmulated*, uses symbolic execution to emulate the program and capture a more accurate CFG at the cost of performance. Since we prioritize accuracy, we choose *CFGEmulated* to recover the CFG for our analysis. An example of a small section of a CFG extracted using Angr can be seen in Figure 5.1

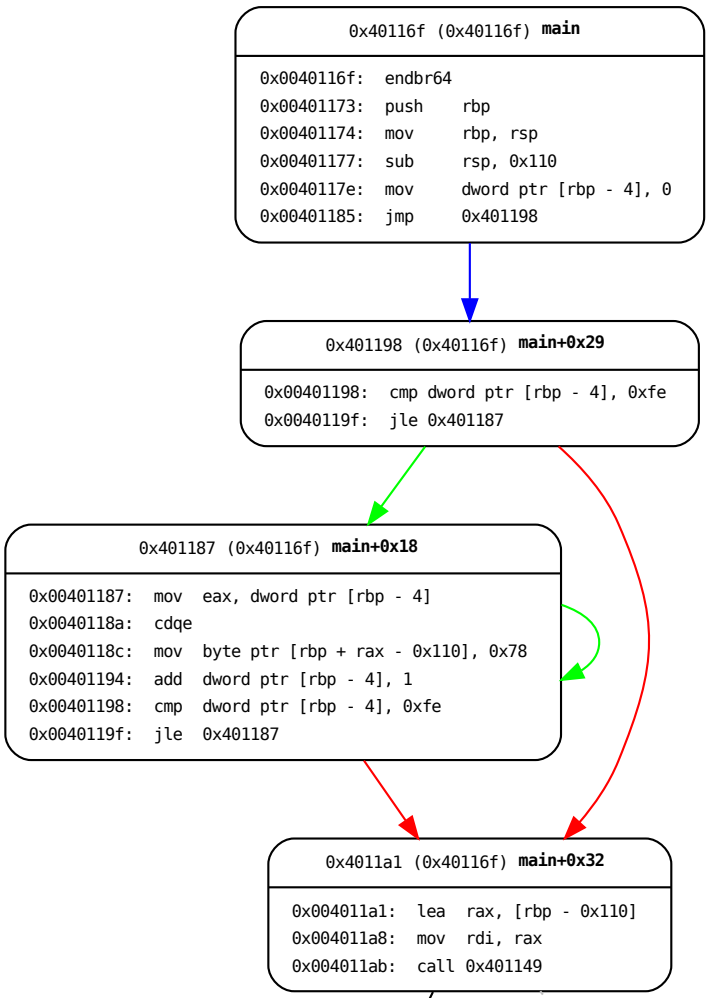

Figure 5.1: Small section of the Control Flow Graph obtained from the assembled C program of Listing 2.1

5.2.2 User Function Data

In addition to the CFG, we also need to extract information regarding user-defined functions in the program. To achieve this, we utilize the previously recovered CFG and iterate through the basic blocks, extracting their names and addresses. Since each basic block is associated with a user-defined function, we can identify all the functions using this method. After identifying the functions, we create a map of the function starting addresses to their names, an example of this map can be viewed in Listing 5.2.

Listing 5.2: Example of a function name map

```
{
    "main" : 0x40123,
    "function_a" : 0x4022d
}
```

5.3 Model Checker

In this section, we will delve into the design and implementation of the model for binary code and the model checker itself, the main component of BASICS. Its function is to create a state space of the stack of the binary program and to verify LTL properties against this state space. Although we will mention LTL properties in this section, the detailed definition of these security properties will be discussed later, after the stack memory states have been properly fleshed out.

5.3.1 Theoretical Stack Memory Model

For our novel model checking approach for binary programs [22], we needed to develop a model appropriate for our model checker. Since we are focusing on stack-based overflow vulnerabilities, it is evident that our theoretical model must represent this region of memory with sufficient detail. For this, we define a model as a Labeled Transition System, where each state is designated as a memory state, composed of stack frames, and the transitions are defined and labeled by memory transition operators:

Definition 5.1 (Stack Memory State Space)**.** *A stack memory state space can be defined as a tuple* $(S, \Gamma, \mathcal{T})$ *where:*

- S *is a set of memory states.*
- Γ *is a set of labels representing memory transition operators.*
- $\mathcal{T} \subseteq S \times \Gamma \times S$ *is the labeled transition relation.*

As an illustration of our transition system, we can consider two memory states M_1, and M_2, we can say that there is a transition from M_1 to M_2, with label `"call function"` if and only if $(M_1, \texttt{"call function"}, M_2) \in \mathcal{T}$, and we can represent it as:

$$M_1 \xrightarrow{\texttt{call function}} M_2$$

In our approach, we define a state of the program's memory as a collection of *function stack frames*. Specifically, at any given point in the program's execution, there exists a set of active stack structures, each represented by a stack frame model of a user-defined function.

Definition 5.2 (Memory State)**.** *A Memory State* $M \in S$ *can be defined as a finite set of active stack frames,* $M = \{F_i, F_{i+1}, ...\}$ *in a given instance of the program execution.*

Furthermore, we conceptualized a model for the stack frames. Since accuracy was a major concern, the stack frame was designed as an array of byte states, mirroring the size of the program's actual stack frames, thus ensuring a one-to-one correspondence with the real stack. The unique feature of this model is that each element of this stack model is a byte state, reflecting the current state of a byte in the real stack. This design choice allows for a detailed and accurate representation of the stack's state at any given point.

Definition 5.3 (Stack Frame)**.** *A Stack Frame $F \in M$ can be defined as a 3-tuple $(B_S, \mathcal{L}, \Sigma)$ where:*

- *B_S: is the finite array of Byte States*
- *$\mathcal{L}$: is the stack frame label*
- *Σ: is the finite set of buffers mapped on the stack frame, where each buffer map $\sigma \in \Sigma$, is defined as $\sigma = \{\texttt{offset}, \texttt{size}\}$*

An example of a Stack Frame with no mapped buffers can be viewed in Figure 5.2

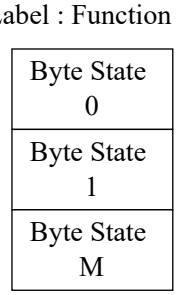

Figure 5.2: Example of a Stack Frame

Each byte in the function stack frame is characterized by one of four states - *Free*, *Critical*, *Occupied*, and *Modified* - as outlined in the automaton in Figure 5.3.

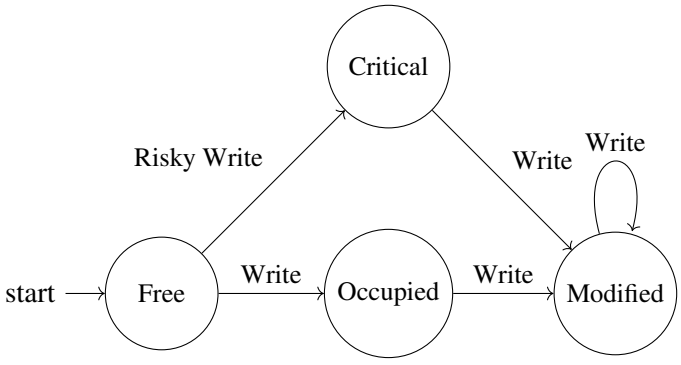

Figure 5.3: Automaton for the Byte States

Transitions between byte states are exclusively triggered by write operations to the stack frame, which are classified as either *risky* or *non-risky*. A *risky* write operation typically occurs when sensitive data, such as return addresses or security tokens, are written to the stack, causing a transition to the *Critical* state. Bytes in this state have an increased vulnerability risk. *Non-risky* writes, on the other hand, transition a byte to the *Occupied* or *Modified* state, depending on the previous state and the type of write operation. The *Free* state signifies unoccupied areas of the stack, which are less likely to be targets of exploitation.

5.3.2 Memory Transition Operators

So far, we have mathematically defined a memory state and its components. However, to generate the memory state space, we need to define the transitions between these states. Although we have introduced *Risky Write* and *Non-Risky Write* transitions, the actual transitions are more intricate and complete. We categorize possible memory transitions into two types: *direct* and *indirect*.

Direct transitions result from a single assembly instruction directly altering the stack frame. For example, instructions like `mov` can directly modify the stack frame. In contrast, *Indirect transitions* arise from function calls that modify the stack frame indirectly. An instance of this is a call to the `strcpy` function, where the effect on the stack is a consequence of the function's execution rather than a direct instruction.

To develop the most accurate model of the program's memory, it was crucial to account for the most commonly occurring write operations in x86-64 Assembly. This necessitated an exhaustive examination of the instruction set in order to identify which instructions have the potential to modify the stack frame. We performed this study and compiled our findings in Table 5.1.

Type of Transition	Operation
Direct	MOV
Direct	PUSH
Direct	POP
Direct	XCHG
Direct	SUB
Direct	LEA
Direct	ENDBR64
Indirect	CALL

Table 5.1: Direct and Indirect Memory Operations

We found that these transitions account for most of the assembly instructions that interact with the stack memory in regular binaries. Additional instructions that interact with the stack were identified, but these were determined to be variations of the instructions present in Table 5.1, such as `cmovz`. Note that despite the instruction `endbr64` being categorized as a direct transition, in reality does not change the stack, but rather it indicates the start of a new stack frame, therefore we consider it as directly affecting the memory state.

The reason for distinguishing the operations into two types was to handle them differently. Since indirect transitions result from the execution of a function, their impact on the stack frame cannot be directly determined as some effects are only detectable during runtime. Therefore, we simulate their effects through concolic execution. In contrast, direct transitions have a predetermined behavior, allowing us to directly calculate their effects on the stack frame.

For direct transitions, we further classify them based on the types of operations they perform on the stack. We identified the main changes that occur in a stack frame and mapped them to the corresponding instructions, compiling the results in Table 5.2. Each of these operations will also

be explained in detail in the following subsections.

Instruction	Operation Type
push	Push
pop	Pop
mov, xchg	Write
sub	Frame extension
endbr64	Frame Allocation

Table 5.2: Mapping of x86-64 to Operation Types

Push

A Push Operation directly affects a single stack frame by appending new bytes to the top of the stack. The number of bytes appended depends on the size of the operand. This operation is exemplified in Figure 5.4.

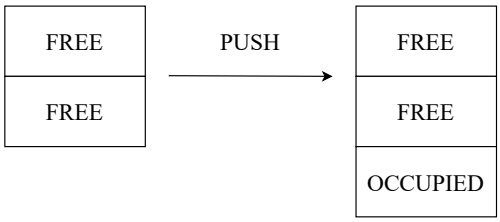

Figure 5.4: Example of a Push operation of Non-Critical data

In general, the byte states appended to the stack frame will be in the state *Occupied*, unless the operand of the instruction is a register containing critical data. For example, `push rbp` saves the stack base pointer to the stack, so the bytes appended would be in the *Critical* state.

Pop

A Pop Operation does the opposite of the push instruction; it directly affects a single stack frame by removing bytes from the top of the stack. This operation is exemplified in Figure 5.5.

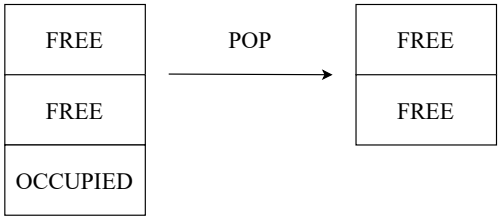

Figure 5.5: Example of a Pop operation

Write

A Write Operation modifies byte states on the stack frame at an arbitrary position. It directly affects a single or a set of byte states by performing a write operation on these and transitioning them to the next state in the automaton 5.3. This operation is exemplified in figure 5.6

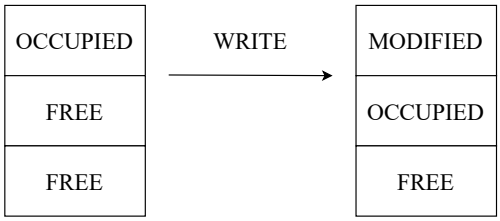

Figure 5.6: Example of a Write operation

Frame Extension

A Frame Extension Operation modifies a stack frame by increasing its size according to the operand. The stack frame is modified by being directly appended with bytes in the *Free* state, matching the offset in the operand. This operation is exemplified in Figure 5.7.

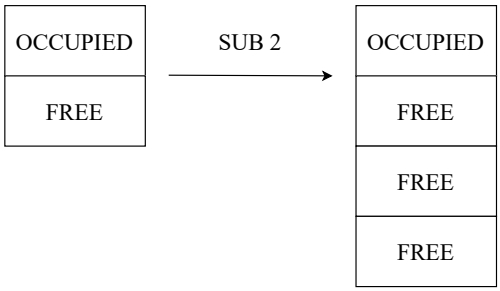

Figure 5.7: Example of a Sub operation

Frame Allocation

Finally, the Frame Allocation Operation modifies a memory state by appending a new stack frame to the set of active stack frames and setting the first 8 bytes of the stack frame in the *Critical* state. These bytes represent the instruction pointer for the callee function and are set up automatically when a call to a user function is performed.

5.3.3 Generating the State Space

To generate the previously defined state space, we began by implementing the theoretical models using the Python Programming Language. In particular, we utilized the library Numpy [26] to build the memory states. Performance was a major concern for the development of the Model Checker, so we chose Numpy as a solution to build our memory states, especially for allocating

arrays for the Stack Frames. Since these need to be as memory efficient as possible, and given that a state space for a medium-sized binary can contain thousands of memory state, any extra bytes per stack frame will quickly add up.

After implementing the Memory States and the Memory Operators, we could finally begin the process of constructing the state space from a disassembled binary. For this purpose, we implemented a Depth First Search (DFS) algorithm on the Control Flow Graph (CFG).

First, we identify the entry point of the binary, typically the `main` function. We then retrieve the corresponding node from the CFG using the address of the entry point. From this node, we extract the first instruction and create an initial memory state with it.

Then to generate the state space we iterate through the basic blocks of the CFG and process one by matching the instructions against the previously defined Memory Transition Operators, for each match found we create a new memory state with the corresponding memory operations applied to the stack frames, and create a transition between the current state and this newly created one, labeling it with the correct Transition Operator.

To ensure the CFG is correctly explored and the generated state space respects the proper control flow of the program, we maintain a stack through the exploration. This stack manages paths and their respective states, with each element being a tuple consisting of a path (a list of basic blocks) and the current state. By keeping track of this stack, we are able to explore different execution paths and maintain the context of each one efficiently.

Our DFS algorithm initiates by creating this stack with the entry point's basic block and the initial state. Then it enters a loop where it processes each path and state in the stack. For each path, it pops the last element, giving us the current path and state. The current basic block it explores is the last one in the path obtained from the stack.

In scenarios where the current basic block is part of a loop, and a maximum iteration count is specified, the block is processed separately in order to simulate the possible effects of the loop. This ensures that infinite loops are executed to at least an upper bound, which can be specified by the user, ensuring a greater degree of accuracy and preventing state space explosion issues.

Next, we retrieve the successors of the current basic block from the CFG, and for each successor, if it is not already in the current path (to avoid cycles), a new path is created by appending the successor to the current path. The new path and the updated state are then added to the stack for further exploration. The full exploration and generation process is described in Algorithm 1.

The final state space obtained is a transition system as described by Definition 5.1, implemented using the Rustworkx library [60]. Rustworkx is a graph library for Python built on Rust, a high-performance programming language. We chose this library due to performance concerns, as we needed to handle state spaces with thousands of states, potentially occupying many gigabytes of memory.

Algorithm 1 Explore State Space

Input: CFG, Entry Point
Output: State Space

$entry\_point \leftarrow$ CFG.get_symbol("main")
$node \leftarrow$ CFG.get_node(entry_point.address)
$entry\_instruction \leftarrow node.instruction[0]$
$initial\_state \leftarrow$ MemoryState(instruction=entry_instruction)
$stack \leftarrow [([node], initial\_state)]$
while $stack$ is not empty **do**
    $path, current\_state \leftarrow stack.pop()$
    $current\_node \leftarrow path[-1]$
    $current\_state \leftarrow$ process_node($current\_node, current\_state$)
    if is_in_loop($current\_node$) and loop_iterations > 0 **then**
        $current\_state \leftarrow$ process_loop($current\_node, current\_state$)
    end if
    $successors \leftarrow$ CFG.get_successors($current\_node$)
    for each $successor$ in $successors$ **do**
        if $successor$ not in $path$ **then**
            $new\_path \leftarrow path + [successor]$
            $stack.append((new\_path, current\_state))$
        end if
    end for
end while

State Space Example

To demonstrate our algorithm and model in action, let's consider the small C program present in Listing 5.3.

Listing 5.3: Small example C program

```
1  int main() {
2      int b = 1;
3      char buf[1];
4      b = 10;
5      buf[0] = '\0';
6      return 0;
7  }
```

By compiling the program in Listing 5.3 and passing as input to our model checker with the flag `--draw-state-space`, we are able to obtain the State Space present in Figure 5.8.

The state space obtained shows the allocation of critical bytes for the return address (RIP) and the push operation that saves the previous stack base pointer onto the stack, pushing another 8 critical bytes. It also shows the proper allocation of memory for the variables `int b` and `char buf[1]`, alongside the modification of the value of `b`.

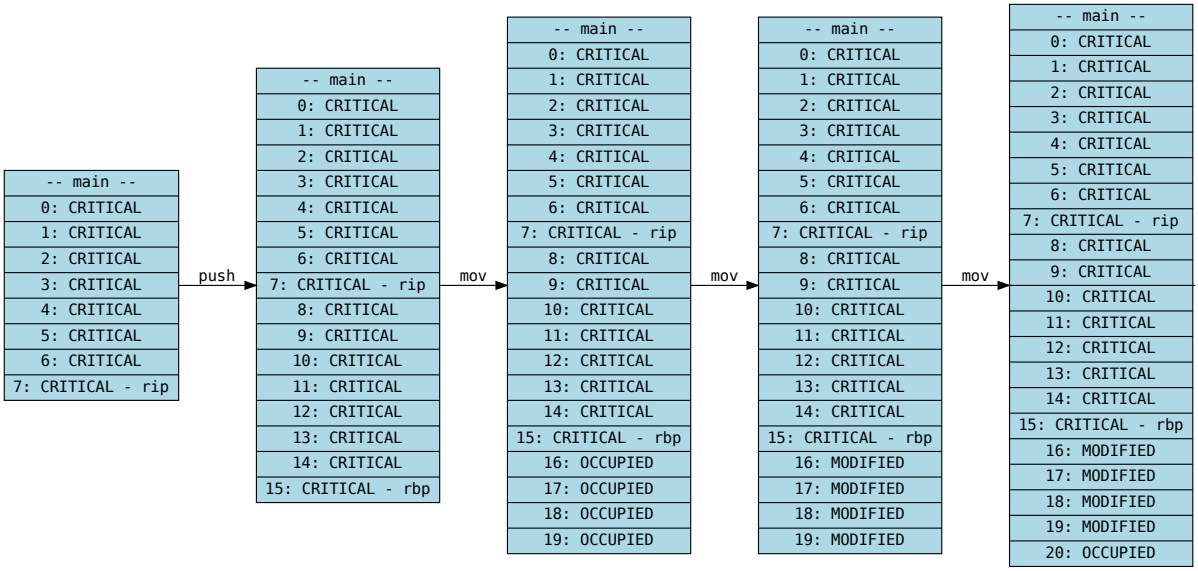

Figure 5.8: Example of the State Space generated for the code in Listing 5.3 automatically using the flag `--draw-state-space` in BASICS

5.3.4 Simulating Calls and Loops through Concolic Execution

So far, we have discussed how we propagate the effects of direct operations onto the memory state, but we have glossed over the more intricate details of indirect operations and loops.

Since the results of these operations can often only be determined at runtime, we perform concolic execution to obtain their results. To achieve this, we utilize the symbolic execution engine of Angr to simulate the calls of functions from the C Standard Library and loops.

Starting with the calls, we distinguish between calls to user-defined functions and calls to C library functions. For C library functions, we redirect the analysis to a call emulator. This emulator begins by constructing a call state by matching the function name to a database of C library functions to determine the number and type of arguments. With this information, we perform a reverse flow analysis of the register values for the argument registers in the basic block containing the call instruction. This process is exemplified in Figure 5.9, where we determine the values for the registers RDI (blue circle) and RSI (green circle), which are the first two registers used to pass arguments in the System V ABI. The identification of these arguments allows us to determine if any buffers passed as arguments exist on the current stack frame, an important detail for the concolic execution process.

After determining this information, we begin the emulation process. First, we determine the address of the current block's function entry point, as each basic block is associated with a user-defined function. To provide context for the symbolic execution, we start our simulation at the beginning of the current user function.

Next, we create a new entry state in Angr using the project factory, which allows us to create program states for any valid address in the program. We create this state with symbolic memory and symbolic registers and set it to the address of the user function's entry point. We then set the

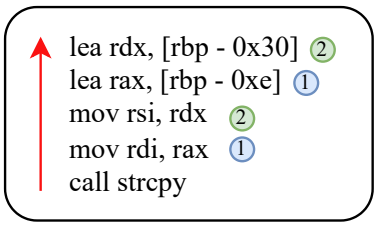

Figure 5.9: Reverse Flow Analysis of a Basic Black containing a Call.

address of the instruction where the call to the C library function occurs as the target and begin the concolic execution process using Angr's explore feature. This feature automatically performs concolic execution from our starting state until it finds the target state. This process is illustrated in Figure 5.10.

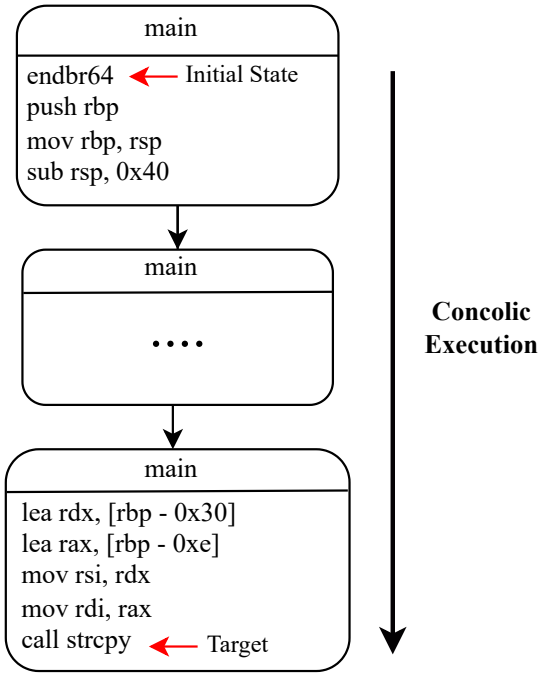

Figure 5.10: Emulation of a function call through Concolic Execution

Angr steps through all the instructions until the target address, stopping just before executing it. To emulate the call, we manually step into it, but before doing so, we concretize and save the contents of the current function's stack frame. After stepping into the call and allowing Angr to emulate it, we concretize and save the stack contents again.

By comparing the stack contents before and after the call, we observe the changes made by the call. We record the positions of the changed bytes, allowing us to propagate the writes to the stack frame accurately. Additionally, for functions that take input from stdin, we extract the concretized bytes that were changed, effectively obtaining the concrete input determined by Angr

through concolic execution. We save this input for later use.

After performing concolic execution on a function call, we hook the address of the instruction where the call occurred and set it to be ignored by Angr in future executions to avoid state space explosion issues. If a function fails to emulate due to existing hooks, we reattempt the emulation without the hooks. If Angr completely fails to emulate the function, we assume no stack changes occurred. This best-effort approach ensures we generate a state space, even if slightly less accurate, rather than no state space at all.

Loops

For loops, the process is very similar to calls. We start by identifying the loop entry and exit points through an Angr analysis that identifies existing loops within a CFG. We create an initial state at the entry point of the current user function and set a target address for the loop's entry point, beginning concolic execution until we reach the entry point. Unlike function calls, we cannot simply step into the loop; we must set a new target address, the loop exit address, and step through the assembly instructions until we reach that address. To handle potentially infinite loops, we set a maximum number of iterations and count each time we return to the loop entry address. Once we reach this number or the exit address, we break out of the loop.

To determine the effects of the loop on the stack, we save the stack contents when we first reach the entry address and after breaking out of the loop. By comparing both stacks, we identify which bytes were affected.

5.3.5 Verifying LTL Properties

Finally, after obtaining a memory state space, the final step in our model checking process is verifying properties against it. We previously defined that our security properties are written as LTL formulas, and that these formulas could be translated to omega automata to allow their verification against a state transition system. Before we discuss the syntax of these formulas, or even how they are translated, let's look at the process of verifying a formula expressed as an omega automaton against our state space.

The algorithm for this is straightforward: we compute the product of the state space and the automaton and look for accepting runs. To do this, we first take an omega automaton, e.g., see Figure 5.11.

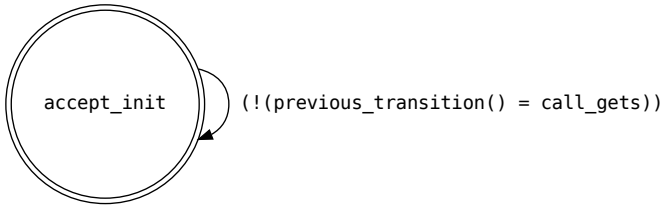

Figure 5.11: Omega automaton for the security property "No Gets Usage" A.4

Then, we perform a Breadth-First Search (BFS) through the generated memory state space

alongside the automaton. For each state transition, we evaluate if the current state satisfies any of the current conditions for the transitions in the automaton. If so, we transition both the state space and the automaton to the next state. If we reach a state where no valid transition exists in the automaton or the automaton is in a non-accepting state at the end of a branch, we consider that branch as non-conforming to the property, and therefore that the binary does not satisfy that security property.

The following is the model checking algorithm implemented to verify the LTL properties expressed as omega automaton:

Algorithm 2 Model Checking Algorithm for ω-automaton

Input: State Space, Omega Automaton
Output: Counter-Example Trace

Initialize *queue*
Initialize a set for visited states
Enqueue the initial state (state_space initial state, omega_automaton initial state, empty trace)
while *queue* is not empty **do**
    $(current\_state, automaton\_state, trace) \leftarrow$ dequeue(queue)
    if $(current\_state, automaton\_state)$ in visited **then**
        continue
    end if
    Add $(current\_state, automaton\_state)$ to visited
    Add $current\_state$ to $trace$
    $violation \leftarrow True$
    for each $transition$ in omega_automaton.get_transitions(automaton_state) **do**
        if current_state.satisfies($transition$) **then**
            $violation \leftarrow False$
            $next\_transition \leftarrow transition$
            break
        end if
    end for
    if $violation$ **then**
        return $trace$
    end if
    for each $successor$ in state_space.get_successors($current\_state$) **do**
        Enqueue $(successor, next\_transition, trace)$
    end for
end while

When a violation is detected for a property, the model checker emits the current trace as a counter-example trace. This trace consists of the sequence of assembly instructions that led the program to that invalid state.

At the end of the model checking process, a report is generated. This report includes all the properties that were verified and all the counter-examples for the properties that were found to be violated. See Listing 5.4 for an example of such a report.

Listing 5.4: Example of a Model Checking Report

```
Verified Security Properties:
  - no_off_by_one_overflow
  - no_underflow_clib
  - canary_integrity
  - no_suspect_overflows
  - no_underflow_loops
  - rip_integrity
  - no_gets_usage

Found security property violations:

Property: rbp_integrity

Counterexample Trace:
0x4011ca:    endbr64
0x4011ce:    push    rbp
0x4011d2:    sub    rsp, 0x10
0x4011d6:    mov    dword ptr [rbp - 4], edi
0x4011d9:    mov    qword ptr [rbp - 0x10], rsi
0x40121d:    call    0x401189
0x40118d:    push    rbp
0x401191:    sub    rsp, 0x50
0x401195:    mov    qword ptr [rbp - 0x48], rdi
0x40119d:    lea    rax, [rbp - 0x40]
0x4011a7:    call    0x401070

1 security property violations found.
```

5.4 Translating LTL Security Properties

The process of model checking a binary has been explained in detail, but now we must discuss how these security properties are specified, and how they are translated to ω-automaton.

This module of our solution, allows users to specify their own properties via LTL formulas, and verify then on binary programs. We developed our own operators that allow to reason about bytes, stacks, buffers and calls in our memory model. With this, we hope that in the future by following this approach, more complex behaviors such as Return-Oriented Programming (ROP) and other unitended behaviors that might not necesseraly classify as malicious, are able to be detected in binaries.

5.4.1 Security Property Specification

Starting with the security properties, these are LTL formulas as previously mentioned, that can be specified by the user for the model checker to verify. They follow the LTL syntax of the tool LTL2BA [23], which can be found in Definition 5.4.

Definition 5.4 (Linear Temporal Logic Syntax)**.** *The syntax of the LTL formulas is defined as follows:*

Propositional Symbols:

- *true, false*
- *Any lowercase string*

Boolean operators:

- `!` *(negation)*
- `->` *(implication)*
- `<->` *(equivalence)*
- `&&` *(and)*
- `||` *(or)*

Temporal operators:

- `[]` *(always)*
- `<>` *(eventually)*
- `U` *(until)*
- `R` *(release)*
- `X` *(next)*

Memory Model LTL Operators

To better reason about our memory model, we define additional operators that allow users to directly specify conditions for arbitrary bytes, stack frames, and buffers, as well as obtain information about previous transitions and the existence of canaries in a given stack frame.

Definition 5.5. *Stack*(f)*: Given a function* f*,* Stack(f) *denotes the stack frame allocated for* f*.*

Definition 5.6. *Byte*(s, i)*: For a stack frame* s*,* Byte(i, s) *returns the current state of the byte at position* i *within* s*.*

Definition 5.7. *Buffer*(s, b)*: For a stack frame* s *and a buffer* b*, returns the size of the buffer* b*.*

Definition 5.8. *Start*(b)*: For a buffer* b*, returns the position of the first byte of the buffer* b*.*

Definition 5.9. *Previous_Transition: Returns a string representation of the previous state transition.*

Definition 5.10. *Has_Canary*(s)*: For a stack frame* s*, returns* `True` *if* s *contains a canary.*

Quantifiers

Additionally, we define syntax for the operators $\forall$ (forall) and $\exists$ (exists) to reason about all the existing stack frames and buffers, without prior knowledge of their existence.

Definition 5.11. *forall_{stack, buffer}: Performs a logical conjunction for a given proposition across all existing stack frames or buffers.*

Definition 5.12. *exists_{stack, buffer}: Performs a logical disjunction for a given proposition across all existing stack frames or buffers.*

5.4.2 Modeling Vulnerabilities with Security Properties

Detecting vulnerabilities is one of the main purposes of our novel approach, for this, we created this entire model-checking framework, and we can now finally discuss how to model vulnerabilities with security properties.

Since we are mainly concerned with buffer overflows, we can start by specifying what happens in the work case when a buffer overflow occurs, i.e. the stack base pointer address and the instruction pointer for the previous function are overwritten. In our model, these sections of the stack should always contain bytes with the state *Critical*, as they should never be modified. With this information, we can define the first two and arguably the most important security properties for the stack memory, that neither the return address nor the stack base pointer should be modified.

To define this property for the return address, we can write that it should be true for every memory state that for all stacks in that state, the first 8 bytes should all have their state equal to *Critical*:

$$\square \left(\forall_s \in \Sigma \left(\bigwedge_{i=0}^{7} \text{byte}(i, \text{stack}(s)) = Critical \right) \right) \tag{5.1}$$

Similarly, for the stack base pointer, we can write the same thing, but instead for the bytes between positions 8 and 15 on the stack:

$$\square \left(\forall_s \in \Sigma \left(\bigwedge_{i=8}^{15} \text{byte}(i, \text{stack}(s)) = Critical \right) \right) \tag{5.2}$$

Where Σ is the set of all stack frames in a given memory state.

The resulting formulas may be simple but, they are very effective at detecting destructive buffer overflows. For a more complicated behavior, we can consider detecting underflows caused by loops. To detect these behaviors, we must consider what would happen to our memory model if one of these vulnerabilities were to occur.

To begin, as the name implies, this type of underflow can only happen if a loop transition occurred before it. After this transition, there should eventually be a simultaneous write operation to the byte positioned at the start of the buffer and the byte directly below it, i.e., with an offset of

1 from the start of the buffer, changing their state to *Occupied*. Additionally, it should be verified that the byte at an offset of 2 from the start of the buffer was not changed to *Occupied*.

In other words, the loop caused a write to `buffer[0]` and `buffer[-1]`, where `buffer[-1]` was previously *Free*, and `buffer[-2]` is not *Occupied*. This indicates that there was no variable stored in the addresses immediately above the buffer, and the loop operation wrote an extra byte outside the buffer, see Figure 5.12. We have these rather complex conditions to avoid false positives. Often, variables will be stored at the addresses immediately above the buffer, and in such cases, we cannot determine if the modification to those bytes was due to the buffer underflow or if a simple move operation occurred during the loop emulation, something that happens frequently.

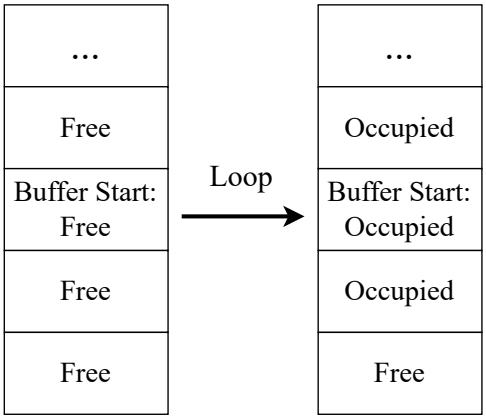

Figure 5.12: Example of a Underflow due to a Loop represented in the Memory Model

To write this into a security property, we can specify that, in all states, it must not occur that the previous transition is a loop, until and including the state (release), in which eventually, there exists a stack that contains any buffer where the starting byte is set to *Occupied*, and the byte directly below it is also set to *Occupied*, while the byte below this one is not *Occupied*. This is expressed in the following LTL formula:

$$\begin{aligned}\Box(\neg(\text{previous\_transition} = \text{loop}\ \mathcal{R}\ \Diamond \\ (\exists_s \in \Sigma(\exists_b \in \beta(\text{byte}(\text{start}(\text{buffer}(\text{b},\text{s})),\text{s}) = \mathit{Occupied}\ \wedge \\ \text{byte}(\text{start}(\text{buffer}(\text{b},\text{s})) + 1,\text{s}) = \mathit{Occupied}\ \wedge \\ \text{byte}(\text{start}(\text{buffer}(\text{b},\text{s})) + 2,\text{s}) \neq \mathit{Occupied}\ \wedge))))\end{aligned} \tag{5.3}$$

Where β is the set of all buffers in a given stack frame.

Besides these security properties, we also created properties to verify the integrity of the stack canaries, to verify that the function `gets` is never used, and additionally to verify that underflows due to the C Library function calls do not occur. These alongside the previously defined security properties are present in Appendix A, and are included by default in the security properties the model checker will attempt to verify for any given binary.

5.4.3 Converting LTL to ω-automaton

As mentioned in Section 5.3.5, to perform the model checking procedure we have to translate these security properties written in LTL to ω-automaton.

There are several algorithms to perform this translation [24, 20], and we could implement one of these to perform this task. However, to save implementation time, we decided to use a pre-existing tool, `LTL2BA` [23], which implements the algorithm used by Spin to perform this conversion. This tool, when given an input LTL formula, outputs an omega automaton in Promela syntax. Since our solution does not work with Promela, we must parse this Promela program to create an automaton using the Rustworkx library.

We will exemplify how the conversion process is executed for the RIP integrity property.

First, LTL2BA is given the path to the security properties directory, where it finds the security properties saved in files with the `.ltl` extension. The contents of this file are shown in Listing 5.5.

Listing 5.5: Contents of rip_integrity.ltl

```
[] ($forall_stack x: (byte(stack(x), 0) = Critical &&
                      byte(stack(x), 1) = Critical &&
                      byte(stack(x), 2) = Critical &&
                      byte(stack(x), 3) = Critical &&
                      byte(stack(x), 4) = Critical &&
                      byte(stack(x), 5) = Critical &&
                      byte(stack(x), 6) = Critical &&
                      byte(stack(x), 7) = Critical)$)
```

Since LTL2BA does not support our defined operators and quantifiers, we place expressions containing these between $ characters. This allows us to extract the expressions and replace them with generic names that LTL2BA can handle, which are later replaced again for the final automaton.

After processing the formula, LTL2BA outputs the automaton in the format of a never claim in Promela, as shown in Listing 5.6.

Listing 5.6: Never claim for the RIP integrity property

```
never { /*  [] (p_0)  */
accept_init:
        if
        :: (p_0) -> goto accept_init
        fi;
}
```

We then parse this never claim using a lexer we developed, building a simple Abstract Syntax Tree (AST). In this AST, we replace the generic expressions with their corresponding propositions extracted from the `.ltl` file. For example, in Listing 5.6, `p_0` is replaced with the quantifier

expression present in the original LTL formula. Finally, we iterate through the AST and build a corresponding omega automaton with the Rustworkx library. The final result for the RIP integrity property is shown in Figure 5.13.

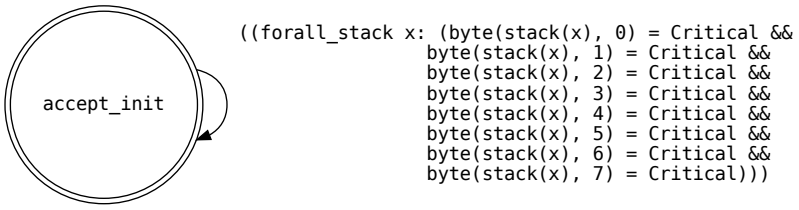

Figure 5.13: Final Omega Automata obtained for the RIP Integrity Security Property

5.5 Identifying and Patching Vulnerabilities

We have explained how security properties can be written to model vulnerabilities, and how they can be verified against a memory state of a binary. Now we will discuss, how we correlate these security properties to CWE classes, and how we go about identifying their source and patching them.

5.5.1 Correlating Security Properties to CWE vulnerability classes

First, correlating the security properties to CWE classes was a rather straightforward task. We created a database in JSON, that can be modified by the user if he or she wishes to add new properties that correlate to other CWE classes. To create the existing mappings, we researched the CWE database and identified which classes our security properties represented. Some of the vulnerability mappings we created can be seen in Listing 5.7.

5.5.2 Examining Counter-Example Traces

To determine the source of vulnerabilities, we need to analyze the report emitted by the model checker. As previously mentioned, this report contains Counter-Example Traces for each vulnerability that was found to be violated (see Listing 5.4). These Counter-Example Traces consist of a sequence of instructions that led the program to an invalid state.

To identify the instruction responsible for the vulnerability, we perform a reverse-flow analysis, meaning we analyze this trace backward. Since most of the buffer overflows we have detected are due to function calls, we generally find the offending instruction to be the call instruction present at the end of the trace. However, this is not always the case with loops.

Currently, we only assign vulnerabilities to function call transitions and loop transitions, as we have not developed a method to find more intricate and niche vulnerabilities caused by other sets

Listing 5.7: Security properties mapped to CWE vulnerability classes

```
{
    "property" : "rip_integrity",
    "vulnerability" : ["CWE-121"]
},
{
    "property" : "rbp_integrity",
    "vulnerability" : ["CWE-121"]
},
{
    "property" : "no_off_by_one_underflows_loops",
    "vulnerability" : ["CWE-124"]
},
{
    "property" : "no_gets_usage",
    "vulnerability" : ["CWE-121"]
}
```

of instructions. Since the report is always emitted with counter-example traces, the user is free to analyze these further.

Once a potential vulnerability is identified, we save its address and class, report it to the user, and move to the next stage of patching.

5.5.3 Patching Process

Removing vulnerabilities from binary programs is a complex task. Various open-source tools employ different techniques for this purpose. We chose to work with E9Patch [17] due to its user-friendliness and effective results for both small and large-scale binaries.

E9Patch bypasses target instructions using a trampoline approach, which necessitated defining patch templates containing code to execute in place of the target instructions. This approach allows us to automatically patch calls from the C Library by replacing known vulnerable behavior with non-vulnerable, identical behavior. For other types of instructions, manual analysis is required to develop a custom patch.

The patching process begins by taking the report from the vulnerability identifier and matching the offending calls to our list of patches. If a patch exists for a specific C Library call, it is applied; otherwise, the patching process is skipped. We have implemented patches for the following C Library functions: `strcpy`, `scanf`, `sprintf`, `gets`, and `strcat`. Users can expand the number of patches by including the patch binary in the directory named patches.

Once a match for the patch is found, we use the call state for that function call, which was determined and saved before starting the concolic execution process (described in section 5.3.4). This call state contains information about the function arguments found in the assembly code, often allowing us to accurately determine the size of buffers passed to the function call. We also extract information about the presence of a stack canary.

Based on the information gathered from the call state, we have prepared different patch tem-

plates. For example, the patch for the `strcpy` function replaces it with the safer alternative `strncpy`. The code for this patch is shown in Listing 5.8.

Listing 5.8: Patch template for strcpy function

```
1  #include "stdlib.c"
2
3  void apply_patch(char *rdi, char *rsi, intptr_t size)
4  {
5      strncpy(rdi, rsi, size - 1);
6  }
```

This patch works by taking the addresses of RDI and RSI, which contain pointers to the target and source buffers, respectively, and a size argument for the total size of the target buffer. By extracting information from the call state, we can determine the size of the target buffer and provide all necessary information for the patch.

Suppose we cannot extract information from the call state and, therefore, cannot determine the size of the target buffer. In that case, we use a different patch template that calculates the maximum possible size for the target buffer during runtime. The code for this patch is shown in Listing 5.9.

Listing 5.9: Patch template for strcpy function with unknown rdi buffer size

```
1  #include "stdlib.c"
2
3  void apply_patch(bool canary, void *rbp, char *rdi, char *rsi)
4  {
5      long offset = 0;
6      if (canary)
7      {
8          offset = 8;
9      }
10     long size = (long)rbp - (long)rdi - offset;
11     strncpy(rdi, rsi, size - 1);
12 }
```

This patch requires information about the presence of a canary and the current stack base pointer address. It calculates the size of the target buffer as the distance between the pointer in RDI and the pointer in RBP, minus the offset if a canary exists. Although this version of the patch is less accurate and may result in increased target buffer sizes, it is still effective at preventing the corruption of the stack canary, the previous stack frame base pointer, and the return address.

After identifying the correct patch, arguments, and the address of the offending call instruction, the patcher replaces this information in the command template and calls the E9Patch tool with the command template shown in Listing 5.10.

If the patch process is successful, a binary with `_patched` appended to the file name is saved in the same directory as the original binary.

Listing 5.10: E9Patch Command Example

```
e9tool -M (addr == CALL_ADDRESS) -P replace
↪  apply_patch(PATCH_ARGS)@PATCH_PATH BINARY_PATH -o
↪  BINARY_PATH_patched
```

Additionally, we have developed patches for four other C Library functions, which can be found in Appendix B.

5.5.4 Validating Patches

The final step after successfully patching a binary is to validate whether the patch has effectively fixed the original vulnerability. Initially, our approach involved re-performing the model checking process on the newly patched binary to determine if the previously violated security properties were still being violated post-patch.

However, due to technical limitations, traditional disassemblers, including Angr, cannot detect changes in the disassembled code. This results in the state space generated for the original binary being identical to that of the patched version. Even if the patch is correctly applied and prevents the application from crashing, the state space obtained remains the same. Consequently, we had to rethink our approach in light of this limitation and settled on validating the patches using tests with inputs extracted during concolic execution.

Inputs are extracted when the stack contents are concretized for functions that take input from `stdin`. These inputs are then saved to a file, which our validator reads and extracts. Since we can determine whether the inputs came from `argv` or `stdin`, we can correctly pass them to the program. With these inputs, we can test if an input that previously caused a crash in the binary still causes a crash in the patched version. This method allows us to determine with some accuracy if our patch was successful, however, for cases in which neither the original binary nor the patched version crashes, manual inspection by the user is required. In either case, the validator emits a report containing this information.

Listing 5.11: Report emitted for a successful patch

```
Running example with test input: aaaabaaacaaadaaaeaaaf
Running exmaple_patched with test input: aaaabaaacaaadaaaeaaaf
Original binary crashed, patched binary did not. Patch was successful.
```

Chapter 6

Evaluation

In this chapter, we will detail the procedures used to evaluate the implementation of BASICS. We will begin by discussing the setup utilized for testing, then characterize the dataset used, and finally discuss the results obtained in the subsequent sections.

Given the challenges previously identified in Section 4.1, we have identified several key aspects for evaluating our implementation: the accuracy of the state space, the capability of the model checker to detect property violations, the ability of the security properties to model vulnerabilities, the efficacy of the patches, and the scalability of the implementation. Based on these aspects, we propose the following research questions, which we will address in this evaluation:

- **Q1**: Is the generated memory state space accurate?
- **Q2**: Do the security properties accurately model buffer overflow vulnerabilities?
- **Q3**: Does BASICS detect property violations when vulnerabilities occur?
- **Q4**: Are the performed patches effectively mitigating these vulnerabilities?
- **Q5**: Does BASICS scale for larger binaries?

6.1 Evaluation Setup

To evaluate our tool's capabilities more thoroughly, we divided the evaluation into two parts. Firstly, we used a synthetic dataset composed of small C programs from the NIST SARD repository [45] to assess the tool's detection accuracy and patch efficacy.

These programs were classified as either vulnerable or non-vulnerable, which served as the ground truth for comparison against our tool's results. With this data, we created a confusion matrix and determined the following metrics:

- **True Positives (TP)**: The number of correctly identified vulnerable programs.
- **True Negatives (TN)**: The number of correctly identified non-vulnerable programs.

- **False Positives (FP)**: The number of non-vulnerable programs incorrectly identified as vulnerable.
- **False Negatives (FN)**: The number of vulnerable programs incorrectly identified as non-vulnerable.

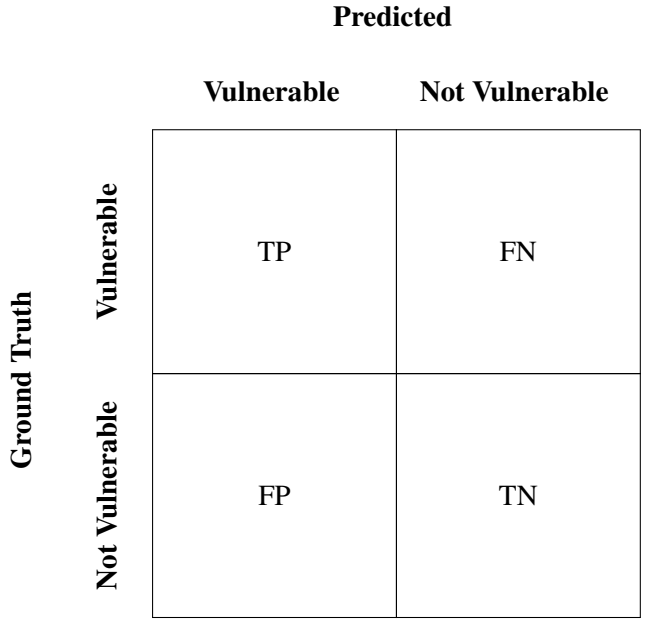

Figure 6.1: Confusion Matrix

Using these metrics, we can then calculate:

- **Accuracy**: The proportion of true results (both true positives and true negatives) among the total number of cases.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.1}$$

- **Precision**: The proportion of true positive results among all positive results.

$$Precision = \frac{TP}{TP + FP} \tag{6.2}$$

- **Recall**: The proportion of true positive results among all actual positives.

$$Recall = \frac{TP}{TP + FN} \tag{6.3}$$

- **F1 Score**: The harmonic mean of precision and recall.

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{6.4}$$

For the second phase, we utilized real open-source applications written in C, sourced from repositories such as SourceForge, GitLab, and GitHub. In this phase, our focus was on testing the scalability of our tool across applications of varying sizes. Therefore, we did not calculate the previous metrics for this evaluation step. Instead, the emphasis was on assessing the tool's performance in terms of handling larger and more complex codebases.

6.1.1 Experimental Setup

All experiments were conducted on an Ubuntu 24.04 virtual machine with 4 CPU cores and 48 GB of RAM. The VM was hosted on a Proxmox server equipped with 2 Xeon E5-2670 processors and 64 GB of DDR3 RAM. The toolchain configuration included Python 3.10.12 implemented via PyPy 7.3.12, Angr 9.2.102, LTL2BA 1.3, and E9Patch 1.0.0-rc9.

6.2 Evaluation with the NIST SARD dataset

In this section, we will detail the dataset used to evaluate the detection capabilities of our tool and the efficacy of the patches. Following this, we will present and discuss the obtained results, addressing research questions Q1-Q4.

6.2.1 Dataset Characterization

The dataset obtained from NIST SARD [45] contains 135 small programs written in the C programming language, with approximately 20 lines of code (LoC) each. These programs include functions such as `strcpy`, `scanf`, `sprintf`, and `gets`, which our tool is capable of patching. Each program was manually classified as either containing an instance of a Buffer Overflow vulnerability or not, to establish a ground truth dataset. In total, the dataset comprises 53 vulnerable cases and 82 non-vulnerable cases. A breakdown of the dataset by function type is presented in Table 6.1.

Function Type	Function	Cases
Input	gets	2
	scanf	29
Data Manipulation	strcpy	79
Output	sprintf	25
Total		135

Table 6.1: Breakdown of the test cases obtained from NIST SARD.

6.2.2 Detection Results

We compiled all these test cases using the GCC compiler with the default flags and processed the resulting binaries with our tool. For each binary, we attempted to verify the following security

properties: *No Gets Usage*, *RIP Integrity*, *RBP Integrity*, *Canary Integrity*, and *No Off-by-One Overflow*. The definitions for each property are provided in Appendix A.

The output of our tool included a model checking report, a visualization of the state space, and, when applicable, a patched binary. We classified a binary as vulnerable if at least one of the security properties was violated, as this would result in a potential CWE-121 report from our tool. We compiled our detection results in a confusion matrix in Table 6.2, and the calculated metrics in Table 6.3

		Tool Classification	
		Vulnerable	**Not Vulnerable**
Ground Truth	**Vulnerable**	34	19
	Not Vulnerable	3	79

Table 6.2: Confusion matrix for the classification results of the NIST SARD dataset.

Metric	**Result**
Accuracy	0.84
Precision	0.92
Recall	0.64
F1-Score	0.76

Table 6.3: Metrics obtained for the classification results of the NIST SARD dataset

Based on the results presented in both Table 6.2 and Table 6.3, we can assert that our tool, leveraging our novel approach, successfully identified vulnerabilities in 64% of all vulnerable cases. Additionally, it generated false positives for only 3 out of 82 non-vulnerable cases, resulting in a precision of 92% and an overall accuracy of 84%.

To better understand the results of our tool, and find the root causes of the 3 instances of false positives and 19 instances of false negatives, we conducted a further analysis of the generated state space for some of these cases.

Starting with the 3 false positives, after examining the state space and model checking reports, we found that our direct memory operators correctly constructed the state space. However, the problem lay in the emulation of the `strcpy` function in all three cases. The resulting stack changes from the concolic execution process indicated a larger write to the stack than in reality, triggering a security property violation in our model checker. This issue seems to stem from our implementation of the `strcpy` function emulation process rather than the state space construction itself. It should be possible to mitigate this issue by revising the implementation of the `strcpy` emulation.

Regarding the 19 false negatives, our examination of the state space revealed two separate issues. The first issue, similar to that of the false positives, was related to the concolic execution of C library functions. In some cases, the process reported no changes in the stack when, in

reality, there were changes, leading to false negatives. This issue could be mitigated by reviewing the concolic execution implementation and gaining a better understanding of how Angr emulates these functions.

The second issue involved overflows not modeled by our current security properties, such as an extra byte being written to the end of a buffer. Listing 6.1 presents such a case. Since we have no properties modeling this kind of behavior, the tool could not detect the presence of a vulnerability in these cases. Addressing this would require implementing the relevant security property. However, this necessitates a careful study of how such behavior affects the state space. If the property is too broad, the tool will produce false positives, and if it is too restrictive, it will not detect this behavior.

Listing 6.1: Example of a buffer overflow not modeled by the defined security properties in Appendix A.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main (int argc, char **argv) {
5      char str1[10];
6      char str2[10];
7      scanf("%11s", str2);
8      strcpy(str1,str2);
9      printf("String copied: %s\n", str1);
10     return(0);
11 }
```

Finally, to fully understand our results, we analyzed the true positives by examining the emitted reports and state spaces alongside the original C code. We concluded that the generated state space was accurate and correctly represented the behavior of the C code. Furthermore, the counter-examples in the reports accurately matched the vulnerable paths in the state space.

With the conclusion of this data analysis, we can answer Q1, Q2, and Q3 from our research questions.

Regarding Q1, we can answer affirmatively. We consider the state spaces generated as accurate since most of them correctly represent the expected memory operations found in the C source code, except for the cases where the concolic execution process yields no results.

For Q2, we find that our defined security properties can model destructive buffer overflows, such as those that overwrite the stack canary, the stack base pointer, or the return address. More subtle overflows, such as writing an extra byte at the end of a buffer, are not modeled by our defined security properties. However, these could also be modeled by defining additional properties. Therefore, we can also answer affirmatively to Q2.

Finally, for Q3, we found that when the state space was accurately constructed, the model checker was able to detect violations of security properties that modeled stack buffer overflows. Consequently, it was able to detect the presence of these vulnerabilities in the program, allowing us to answer Q3 affirmatively.

6.2.3 Patching Results

In addition to detecting vulnerabilities, our tool also attempts to fix them via corrective patches. Although we have implemented corrective patches for only a few C library functions, the dataset we are using contains vulnerabilities originating exclusively from these functions. This allows us to evaluate the efficacy of the patching process for the programs classified as vulnerable by our tool.

As previously mentioned, our tool validates the patches using concolic inputs extracted from the concolic execution process. For each case, it emits a report stating whether the program crashed before and after the application of the patch. If the patch prevents the program from crashing, we report the patch as successful. Additionally, if a program did not crash before and after the patch, we also consider the patch successful but issue a warning for the user to manually check the patch. Considering this, we broke down the results from the reports by the function being patched and compiled them in Table 6.4.

	Positive Cases		**Patch**	
Function	**True**	**Reported**	**Perfomed**	**Successful**
strcpy	23	26	26	26
sprintf	10	10	10	10
gets	1	1	1	1
Total	34	37	37	37

Table 6.4: Breakdown of the patches performed per function.

The results show that 100% of the patches performed were successful, and the 3 false positives reported were also patched. This raises some questions regarding our patch validation process, so we manually examined each patched binary.

Starting with the 3 reported false positives, we found that despite being patched, the behavior of the programs did not change and continued as intended. We conclude that this was due to our process of determining the arguments for the function calls. Since we can detect these arguments with some degree of accuracy, the behavior of the program does not change when the patch is performed and the function call is replaced by an equivalent one.

For the other cases, we found that the vulnerabilities were successfully patched in all instances. However, we discovered that the behavior of some programs containing `sprintf` was slightly altered. In these cases, the output of this function was different because the patch failed to correctly determine the format string arguments used. Therefore, we conclude that while the patches were effective at removing the buffer overflow vulnerabilities, they did not always preserve the correct behavior. We believe this issue can be corrected by further refining the patch templates. This allows us to answer positively to Q4.

6.3 Evaluation with Open-Source Applications

The dataset from NIST SARD allowed us to evaluate the performance of our tool using synthetic tests. However, these tests might not accurately represent real-world applications. Therefore, to better evaluate our tool, we gathered six applications written in C from open-source repositories such as SourceForge, GitHub, and GitLab. These applications span different contexts, including networks, systems, and an educational project.

The size of the projects varied between 15 to 261 lines of code (LoC), allowing us to observe how our tool scales with differently sized binaries. For these programs, it was unknown whether they had buffer overflow vulnerabilities before testing.

6.3.1 Results

For each of these projects, we compiled them using their respective Makefile when one was present, otherwise, we utilized GCC with the default compilation flags. We then analyzed the binaries with our tool and compiled information about each project, along with our findings when testing the tool, in Table 6.5.

Application	Files	LoC	Verification Time (in seconds)	Potential Vulnerabilities	Patch Performed
Macgen	1	15	10.10	0	0
HTML Parser	1	70	32.64	1	1
IPV6 Validator	1	34	3.46	1	1
Thread-Fifo	3	261	4.79	0	0
Hash-Map	2	203	759.73	0	0
Contacts Management	1	112	188.42	1	1

Table 6.5: Evaluation Results for Open-Source Applications

Starting with the vulnerabilities detected, our tool found potential buffer overflows in three of the six projects: *HTML Parser*, *IPV6 Validator*, and *Contacts Management*. Since these projects were not extensive, we analyzed the source code files and confirmed that they were indeed vulnerable. In both *HTML Parser* and *IPV6 Validator*, we found a misuse of the `strcpy` function, which our tool successfully patched and validated. However, in the *Contacts Management* application, the buffer overflow vulnerability was due to a `scanf` call without a format string. The patch applied by our tool for this program was unsuccessful, causing it to hang.

For the other projects, we manually analyzed the source code and found no obvious buffer overflow vulnerabilities, affirming that our tool correctly classified these as non-vulnerable.

6.3.2 Performance Evaluation

Regarding the performance of our tool, we measured the time it took to build the state space and perform the verification process. The results are compiled in Table 6.5.

Analyzing these results, one might expect the verification time to be proportional to the lines of code (LoC), but this was not the case. For example, the project with the largest codebase, *Thread-Fifo*, had a verification time of 4.79 seconds, significantly faster than the smallest project, *Macgen*, which took 10.10 seconds to verify.

To understand the cause, we inspected the source code and found that while *Macgen* had a single `for` loop, *Thread-Fifo* contained no loops or branches, resulting in a significantly shorter verification time. Similarly, *Hash-Map* and *Contacts Management* were outliers in terms of verification time, taking significantly longer than the other projects. This was due to the number of branches and loops in their code. The *Contacts Management* project had many branches due to the use of `switch` statements and `if` statements within `for` loops. For *Hash-Map*, there were nested loops, which greatly increased the verification time and memory usage.

These results point to the state explosion problem [62], a common issue with formal methods. Despite our efforts to avoid it in our implementation, the number of paths in our state space grows exponentially with the number of branches in a program. For each branch, two paths are created, and for each of these, further paths can be created, and so on. We estimate the number of paths in our memory state space to be proportional to the following:

$$\text{Nº Paths} \propto 2^{\text{Nº Branches}}$$

Thus, for a program with 10 if-else statements, we can expect approximately 1024 possible execution paths in our state space.

Another aspect to consider is the memory required to store the state space. For larger programs ($\sim$ 700 LoC), our main issue with verification was not time but memory, as our tool would fill the 48GB of RAM in our VM and crash the process. We tried several approaches to mitigate this, such as reducing the accuracy of the concolic execution process in the settings of Angr and limiting the simulated functions to those directly affecting the stack, but we ultimately could not circumvent this issue.

Finally, we can answer the last research question, Q5, regarding the scalability of the tool. Considering our results and analysis, we must answer Q5 negatively, as our tool does not scale well for larger binaries due to the state explosion problem.

Chapter 7

Conclusion

This dissertation set out to develop a novel, scalable, and accurate approach to detect and remove stack buffer overflows in binary programs.

To achieve this, we researched several existing techniques for vulnerability detection in binaries, including formal methods. Our research led to the development of a model checking approach to verify security properties in the stack memory of a binary. We implemented this approach in BASICS: Binary Analysis and Stack Integrity Checker System, utilizing the Angr framework to generate the state space and perform concolic execution on function calls and loop constructs. With this generated state space, we verified security properties written in LTL, which modeled buffer overflow vulnerabilities. When these properties were violated, they pointed to potential vulnerabilities. After verifying the properties, any violations identified as addressable vulnerabilities were patched using E9Path to redirect the program control flow to a patch template. We verified the validity of the patches using inputs extracted from the concolic execution process.

With BASICS implemented, we evaluated it using a dataset of vulnerable programs from NIST SARD and real applications collected from SourceForge, GitHub, and GitLab. Our results showed that the tool successfully detected violations of the specified security properties, leading to the detection of buffer overflow vulnerabilities, and performed patches to mitigate some of these vulnerabilities.

The evaluation results showed that our tool had an accuracy of 0.84 in detecting buffer overflow vulnerabilities in the NIST SARD dataset, with a recall of 0.64 and a precision of 0.92. Considering this is a novel approach for detecting these vulnerabilities in binaries through model checking, these results show great promise. We believe that with a better implementation, we could achieve a precision of 1 and an improved recall score. By improving the concolic execution process, gaining a better understanding of the Angr framework and its intricacies, and adding additional security properties to model more subtle vulnerabilities, we could further increase detection accuracy.

For the detected vulnerabilities in the NIST SARD dataset, BASICS managed to perform patches on the programs, as these contained vulnerable calls for which we provided patches. These patches were effective in removing the vulnerabilities but demonstrated some limitations in maintaining the original program behavior, particularly for the `sprintf` function.

An evaluation of BASICS performance was also conducted, but it encountered state explosion

issues, indicating it does not scale well for larger binaries.

Based on the results obtained, we can affirm that we managed to satisfy most of our initial objectives, specifically creating a novel approach to accurately detect buffer overflows. However, we did not succeed in making it scalable as originally intended.

7.1 Limitations

While our approach shows promise, our implementation has some limitations. To better understand these and how they might be addressed, we will go over each one and discuss them in detail:

- **Scalability**: We found our implementation to suffer from the state explosion problem, causing it to fail to scale for larger binaries. This issue plagues model checking approaches, so it was not surprising that our approach would suffer from it as well. The origin of it in our case is the branches present in the assembly code. Since our approach requires all possible paths to be verified, it results in an extremely large number of states. To mitigate this, we could rethink our implementation, particularly if we could do a pre-processing of the binary, where we attempt to find all dead branches and prune them. Another solution would be to simplify the state space and reduce the number of memory operators, but this might lead to inaccuracies in state space.

- **Memory Usage**: During our evaluation process, we discovered our solution used a staggering amount of memory. While this is, of course, directly correlated to the previous issue of scalability, it also has some of its own quirks to consider. Specifically, we found that most of the memory consumed was not from the state space itself, but from Angr, particularly during the concolic execution process. Since we were simulating large chunks of the binary at a time, this was to be expected. In order to mitigate it, we could rethink our concolic execution approach and attempt to simulate smaller chunks of the binary at a time, which should result in improved memory usage and execution time. Another way to decrease memory usage would be to store the differences between each state, instead of storing a separate instance of each one. However, this would not have as drastic of an effect as improving the efficiency of the concolic execution process.

- **Patching Validation**: We utilize inputs extracted during the concolic execution process to validate our patches, and while this process gives us some degree of information about patch efficacy, it often requires further manual inspection, as it is unable to provide more insight than whether the program crashes or not. For this issue, we believe utilizing our initial approach of performing model checking on the patched binary would show more insight into the effectiveness of the patch. However, for this approach, a different patching solution would be required, as with the E9Patch tool, our model checker does not detect changes in the binary.

7.2 Future Work

With the capabilities and limitations of our approach and implementation in mind, we outline several areas for future research to enhance the applicability of our tool:

- **Refining Patch Templates**: Currently, we only address five C-library functions for patching, although users can specify their own patch templates. Future work should focus on improving existing patches to consistently match the original program behavior and adding patches for more complex behaviors, such as underflows and overflows caused by loops.
- **Additional Security Properties**: Our current security properties model only the simplest and most destructive overflow behaviors. Future research should develop security properties that model more complex behaviors, such as ROP and data integrity violations.
- **Incorporating Read States into the Memory Model**: Our current approach models only write operations to the stack. Incorporating read operations would expand our tool beyond buffer overflows, making it a more general stack memory model checker. With these capabilities, we could explore security properties related to data confidentiality.
- **Modeling Heap Memory**: Expanding our tool to include modeling of the heap memory is another area for future research. Since the heap memory also involves its own memory operations, modeling this region alongside the stack memory would enable our tool to check for heap overflows, use-after-free, and double-free vulnerabilities.

7.3 Final Remarks

Formal methods are not usually employed in this area of security, which currently sees most of the recent work published using either dynamic analysis or machine learning. While these methods can yield good results, they lack a critical aspect that we believe is important: proof of absence.

Dynamic analysis can confirm the presence of a vulnerability, but it is unable to confirm its absence. Similarly, machine learning can only provide a classification with a certain probability of being correct. In contrast, formal methods can provide mathematical proof that a certain behavior is absent or present. This capability is extremely powerful, as it allows users to be absolutely certain that their product is free of certain behaviors.

Although this sounds excellent, formal methods face significant challenges, such as the state explosion problem, which can make them impractical for larger problems. Despite this, research in this area is invaluable. By continuing to address and mitigate these issues, we can develop new and improved methods that bring us closer to proving the absence of vulnerabilities in software.

We believe this thesis provides valuable contributions to the field by presenting a new approach that, with better implementation, could become a valuable asset for the analysis of the stack memory in binaries.

Acronyms

BFS Breadth-First search. 43

BMC Bounded Model Checking. 20, 21

CFG Control Flow Graph. 17, 27, 32, 33, 39

CLMs Code Language Models. 22

CPU Central Processing Unit. 10, 16

CWE Common Weakness Enumeration. xiv, xxi, 1, 4, 7, 50, 51

DFS Depth-First Search. 39

ELF Executable and Linkable Format. 17

LoC Lines of Code. 57, 61, 62

LTL Linear Temporal Logic. 12–14, 24, 27, 32, 34, 43, 45, 49, 63

ML Machine Learning. 19

PE Portable Executable. 17

ROP Return-Oriented Programming. 45, 65

Bibliography

[1] CWE - common weakness enumeration. https://cwe.mitre.org/. Accessed: 8-07-2024.

[2] Dennis Andriesse. *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly*. No Starch Press, San Francisco, 1st edition, 2019.

[3] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroSP)*, page 334–349, Paris, April 2017. IEEE.

[4] Andreea Bican, Răzvan Deaconescu, Wei Ngan Chin, and Quang-Trung Ta. Verification of c buffer overflows in c programs. In *2018 17th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–6, 2018.

[5] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Comak, and Leyli Karacay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

[6] El Habib Boudjema, Sergey Verlan, Lynda Mokdad, and Christèle Faure. Vyper: Vulnerability detection in binary code. *SECURITY AND PRIVACY*, 3(2):e100, March 2020.

[7] Muhammad Arif Butt, Zarafshan Ajmal, Zafar Iqbal Khan, Muhammad Idrees, and Yasir Javed. An in-depth survey of bypassing buffer overflow mitigation techniques. *Applied Sciences*, 12(13), 2022.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019.

[9] Hao Chen, Drew Dean, and David A. Wagner. Model checking one million lines of c code. In *Network and Distributed System Security Symposium*, 2004.

[10] Hao Chen and David A. Wagner. Mops: An infrastructure for examining security properties of software. Technical report, USA, 2002.

[11] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 736–747, 2019.

[12] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[13] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer Publishing Company, Incorporated, 1st edition, 2018.

[14] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[15] The U.S. Department. CWE - 2023 CWE top 25 most dangerous software weaknesses. 2023.

[16] W. Du. *Computer Security: A Hands-on Approach*. Wenliang Du, 2019.

[17] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 151–163, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. HeapHopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 99–116, Baltimore, MD, August 2018. USENIX Association.

[19] Eldad Eilam. *Reversing: Secrets of Reverse Enginnering*. Wiley Publishing, Inc., 2005.

[20] Kousha Etessami and Gerard Holzmann. Optimizing büchi automata. volume 1877, 05 2003.

[21] Diogo Ferreira. Automatic binary patching for flaws repairing using static re-writing and reverse dataflow analysis, 2022. http://hdl.handle.net/10451/58214.

[22] Luís Ferreirinha. and Ibéria Medeiros. On the path to buffer overflow detection by model checking the stack of binary programs. In *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, pages 719–726. INSTICC, SciTePress, 2024.

[23] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 53–65, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[24] Rob Gerth, Den Dolech, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. *Proceedings of the 6th Symposium on Logic in Computer Science*, 15, 12 1995.

[25] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Erik Antelman, Alan Mackay, Marc W. McConley, Jeffrey M. Opper, Peter Chin, and Tomo Lazovich. Automated software vulnerability detection with machine learning. (arXiv:1803.04497), August 2018. arXiv:1803.04497 [cs, stat].

[26] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[27] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.

[28] Yikun Hu, Yuanyuan Zhang, and Dawu Gu. Automatically patching vulnerabilities of binary programs via code transfer from correct versions. *IEEE Access*, 7:28170–28184, 2019.

[29] Yao-Wen Huang, Fang Yu, C. Hang, Chung-Hung Tsai, D.T. Lee, and Sy-Yen Kuo. Verifying web applications using bounded model checking. In *International Conference on Dependable Systems and Networks, 2004*, pages 199–208, 2004.

[30] João Inácio and Ibéria Medeiros. Corca: An automatic program repair tool for checking and removing effectively c flaws. *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 71–82, 2023.

[31] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. (arXiv:2302.05020), April 2023. arXiv:2302.05020 [cs].

[32] Arvinder Kaur and Ruchikaa Nayyar. A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science*, 171:2023–2029, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19).

[33] Taddeus Kroes, Koen Koning, Erik Van Der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, Porto Portugal, April 2018. ACM.

[34] laf intel. Circumventing fuzzing roadblocks with compiler transformations, 2016.

[35] K. Rustan M. Leino. *Dafny: An Automatic Program Verifier for Functional Correctness*, volume 6355 of *Lecture Notes in Computer Science*, page 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[36] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1, 12 2018.

[37] Hongyu Liu, Sam Silvestro, Xiaoyin Wang, Lide Duan, and Tongping Liu. CSOD: Context-Sensitive Overflow Detection. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 50–60, Washington, DC, USA, February 2019. IEEE.

[38] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search. In *Proceedings 2023 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2023. Internet Society.

[39] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[40] Iberia Medeiros, Nuno Neves, and Miguel Correia. Statically detecting vulnerabilities by processing programming languages as natural languages. *IEEE Transactions on Reliability*, 71(2):1033–1056, June 2022.

[41] Eric Mercer and Michael Jones. Model checking machine code with the gnu debugger. In Patrice Godefroid, editor, *Model Checking Software*, pages 251–265, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[42] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), jan 2018.

[43] Muhammad Nadeem, Byron Williams, and Edward Allen. High false positive detection of security vulnerabilities: A case study. 03 2012.

[44] Huu-Vu Nguyen and Tayssir Touili. Caret model checking for malware detection. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, page 152–161, New York, NY, USA, 2017. Association for Computing Machinery.

[45] NIST. Software Assurance Reference Dataset (SARD). https://www.nist.gov/itl/ssd/software-quality-group/samate/software-assurance-reference-dataset-sard. Accessed: 20-07-2024.

[46] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7(49), 1996.

[47] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask, 2020.

[48] Eduard Pinconschi, Rui Abreu, and Pedro Adão. A comparative study of automatic program repair techniques for security vulnerabilities. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 196–207, 2021.

[49] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[50] Thomas Reinbacher, Martin Horauer, Bastian Schlich, Jörg Brauer, and Florian Scheuer. Model checking assembly code of an industrial knitting machine. In *Proceedings of the 2009 4th International Conference on Embedded and Multimedia Computing, EM-Com 2009*, 2009.

[51] 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, 2018.

[52] Andreas Schaad and Dominik Binder. *Deep-Learning-Based Vulnerability Detection in Binary Executables*, volume 13877 of *Lecture Notes in Computer Science*, page 453–460. Springer Nature Switzerland, Cham, 2023.

[53] Bastian Schlich and Stefan Kowalewski. [mc]square: A model checker for microcontroller code. In *Proceedings - ISoLA 2006: 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 466–473. IEEE Computer Society, 2006.

[54] B. Schwarz, Hao Chen, D. Wagner, G. Morrison, J. West, J. Lin, and Wei Tu. Model checking an entire linux distribution for security violations. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 10 pp.–22, 2005.

[55] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[56] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, sep 2005.

[57] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 263–272, New York, NY, USA, 2005. Association for Computing Machinery.

[58] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, and Federica Sarro. A survey on machine learning techniques for source code analysis. *ArXiv*, abs/2110.09610, 2021.

[59] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[60] Matthew Treinish, Ivan Carvalho, Georgios Tsilimigkounakis, and Nahum Sá. rustworkx: A high-performance graph library for python. *Journal of Open Source Software*, 7(79):3968, 2022.

[61] 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 *31st USENIX Security Symposium (USENIX Security 22)*, pages 413–430, Boston, MA, August 2022. USENIX Association.

[62] Antti Valmari. *The state explosion problem*, pages 429–528. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[63] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution.

[64] Oualid Zaazaa and Hanan El Bakkali. Dynamic vulnerability detection approaches and tools: State of the Art. In *2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS)*, pages 1–6, Fez, Morocco, October 2020. IEEE.

[65] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, pages 17–36, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[66] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–69, February 2024.

[67] Yang Zhang, Xiaoshan Sun, Yi Deng, Liang Cheng, Shuke Zeng, Yu Fu, and Dengguo Feng. *Improving Accuracy of Static Integer Overflow Detection in Binary*, volume 9404 of *Lecture Notes in Computer Science*, page 247–269. Springer International Publishing, Cham, 2015.

Appendix A

LTL Security Properties

Set of stacks in a given memory state: Σ
Set of buffers in a given stack frame: β

A.1 RIP Integrity

$$\Box\left(\forall_s \in \Sigma\left(\bigwedge_{i=0}^{7} \text{byte}(i, \text{stack}(s)) = Critical\right)\right)$$

A.2 RBP Integrity

$$\Box\left(\forall_s \in \Sigma\left(\bigwedge_{i=8}^{15} \text{byte}(i, \text{stack}(s)) = Critical\right)\right)$$

A.3 Canary Integrity

$$\Box\left(\forall_s \in \Sigma\left(\text{has\_canary(s)} \implies \bigwedge_{i=16}^{23} \text{byte}(i, \text{stack}(s)) = Critical\right)\right)$$

A.4 No gets usage

$$\Box(\text{previous\_transition} \neq \text{call gets})$$

A.5 No off by one overflow

$$\Box(\neg(\exists_s \in \Sigma(\text{byte}(15, \text{stack(s)}) = Modified \wedge \text{byte}(14, \text{stack(s)}) = Critical)))$$

A.6 No underflow clib

$$\Box(\neg(\text{previous\_transition} \in \text{clib}\ \mathcal{R}\ \Diamond\ (\exists_s \in \Sigma(\exists_b \in \beta(\text{byte}(\text{start}(\text{buffer}(\text{b},\text{s})),\text{s}) = \mathit{Occupied}\ \wedge\ \text{byte}(\text{start}(\text{buffer}(\text{b},\text{s})) + 1,\text{s}) = \mathit{Occupied}\ \wedge\ \text{byte}(\text{start}(\text{buffer}(\text{b},\text{s})) + 2,\text{s}) \neq \mathit{Occupied}\ \wedge))))$$

A.7 No underflow loops

$$\Box(\neg(\text{previous\_transition} = \text{loop}\ \mathcal{R}\ \Diamond\ (\exists_s \in \Sigma(\exists_b \in \beta(\text{byte}(\text{start}(\text{buffer}(\text{b},\text{s})),\text{s}) = \mathit{Occupied}\ \wedge\ \text{byte}(\text{start}(\text{buffer}(\text{b},\text{s})) + 1,\text{s}) = \mathit{Occupied}\ \wedge\ \text{byte}(\text{start}(\text{buffer}(\text{b},\text{s})) + 2,\text{s}) \neq \mathit{Occupied}\ \wedge))))$$

Appendix B

Patch Templates

B.1 gets

Listing B.1: Patch template for gets function

```
1  #include "stdlib.c"
2
3  void apply_patch(char* rdi, intptr_t size) {
4      fgets(rdi, size, stdin);
5  }
```

B.2 gets unknown buffer size

Listing B.2: Patch template for gets function with unknown rdi buffer size

```
1  #include "stdlib.c"
2
3  void apply_patch(bool canary, void* rbp, char* rdi) {
4      long offset = 0;
5      if (canary) {
6          offset = 8;
7      }
8      long size = (long)rbp - (long)rdi - offset;
9      fgets(rdi, size, stdin);
10 }
```

B.3 strcpy

Listing B.3: Patch template for strcpy function

```
1  #include "stdlib.c"
2
3  void apply_patch(char *rdi, char *rsi, intptr_t size)
4  {
5      strncpy(rdi, rsi, size - 1);
6  }
```

B.4 strcpy unknown buffer size

Listing B.4: Patch template for strcpy function with unknown rdi buffer size

```
1  #include "stdlib.c"
2
3  void apply_patch(bool canary, void *rbp, char *rdi, char *rsi)
4  {
5      long offset = 0;
6      if (canary)
7      {
8          offset = 8;
9      }
10     long size = (long)rbp - (long)rdi - offset;
11     strncpy(rdi, rsi, size - 1);
12 }
```

B.5 sprintf

Listing B.5: Patch template for sprintf function

```
1  #include "stdlib.c"
2
3  void apply_patch(char* rdi, char* rsi, intptr_t size, ...) {
4      va_list args;
5      va_start(args, size);
6      vsnprintf(rdi, size-1, rsi, args);
7      va_end(args);
8  }
```

B.6 sprintf unknown buffer size

Listing B.6: Patch template for sprintf function with unknown rdi buffer size

```
1  #include "stdlib.c"
2
3  void apply_patch(bool canary, void* rbp, char* rdi, char* rsi, ...) {
4      long offset = 0;
5      if (canary) {
6          offset = 8;
7      }
8      long size = (long)rbp - (long)rdi - offset;
9      va_list args;
10     va_start(args, rsi);
11     vsnprintf(rdi, size-1, rsi, args);
12     va_end(args);
13 }
```

B.7 strcat

Listing B.7: Patch template for strcat function

```
1  #include "stdlib.c"
2
3  void apply_patch(char* rdi, char* rsi, intptr_t size) {
4      strncat(rdi, rsi, size - strlen(rdi) - 1);
5  }
```

B.8 strcat unknown buffer size

Listing B.8: Patch template for strcat function with unknown rdi buffer size

```
1  #include "stdlib.c"
2
3  void apply_patch(bool canary, void* rbp, char* rdi, char* rsi) {
4      long offset = 0;
5      if (canary) {
6          offset = 8;
7      }
8      long size = (long)rbp - (long)rdi - offset;
9      strncat(rdi, rsi, size - strlen(rdi) - 1);
10 }
```

B.9 scanf

Listing B.9: Patch template for scanf function

```
1  #include "stdlib.c"
2
3  void apply_patch(char* rdi, char* rsi, intptr_t size, ...)
4  {
5      char* buf[size];
6      fgets(buf, size-1, stdin);
7
8      va_list args;
9      va_start(args, size);
10     snprintf(rdi, size-1,  rsi, buf);
11     va_end(args);
12 }
```

B.10 scanf unknown buffer size

Listing B.10: Patch template for scanf function with unknown rdi buffer size

```
#include "stdlib.c"

void apply_patch(bool canary, void* rbp, char* rdi, char* rsi, ...)
{
    long offset = 0;
    if (canary)
    {
        offset = 8;
    }
    long size = (long) rbp - (long) rsi - offset;
    char* buf[size];
    fgets(buf, size-1, stdin);
    va_list args;
    va_start(args, rsi);
    snprintf(rdi, size-1,  rsi, buf);
    va_end(args);
}
```