Finding Use-After-Free vulnerabilities using Symbolic Execution with angr

Bachelor's Thesis

by

Lennart Henke
Matr. No.3328068

in fulfillment of requirements for degree
Bachelor of Science (B.Sc.)

submitted to
Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik IV
Arbeitsgruppe für IT-Sicherheit

in degree course
Informatik (B.Sc.)

First Examiner:	Prof. Dr. Peter Martini University of Bonn
Second Examiner:	Prof. Dr. Michael Meier University of Bonn
Supervisor:	Dr. Nils-Edvin Enkelmann University of Bonn

Bonn, May 14, 2023

Statement of Authorship

I hereby confirm that the work presented in this bachelor thesis has been performed and interpreted solely by myself except where explicitly identified to the contrary. I declare that I have used no other sources and aids other than those indicated. This work has not been submitted elsewhere in any other form for the fulfilment of any other degree or qualification.

Bonn, May 14, 2023

L. Henke

Lennart Henke

Acknowledgement

I would like to thank Dr. Nils-Edvin Enkelmann for providing valuable feedback on this work. Furthermore, I want to thank Fraunhofer FKIE for generously providing the necessary computational resources to carry out this research.

Abstract

Use-After-Free (UAF) vulnerabilities are increasingly prevalent in computer systems, highlighting the need for effective detection methods. Although current static binary analysis techniques have successfully identified vulnerabilities, their high false positive rate remains a significant limitation. The thesis introduces a fully automated symbolic execution approach for detecting UAF vulnerabilities in binary code to validate the results from static analysis tools. The proposed method utilizes the `angr` framework for analysis and supports Directed Symbolic Execution, Under-Constrained Symbolic Execution, and Veritesting to mitigate the path explosion problem. The proposed approach is evaluated using the Juliet Test Suite, a widely-used benchmark, and four real-world binaries with known UAF vulnerabilities and CVEs. The analysis successfully detects all UAF vulnerabilities in the Juliet Test Suite without generating false positives. However, it is unable to identify any of the real-world vulnerabilities. These experimental results suggest that, while the proposed approach is highly effective for small-scale test programs, it is not practical for real-world binary analysis. Further improvements are necessary to enhance the method's ability to verify the results of static analysis tools in real-world scenarios.

Contents

1 Introduction

Despite the constant advancements in security measures and testing, the number of Use-After-Free (UAF) vulnerabilities has increased rapidly in recent years [31], which poses a significant threat to the security and privacy of software users. The UAF vulnerability class persists in various real-world programs written in manual memory management languages like C/C++ [27]. This is due to the improper allocation and deallocation of memory, which creates a dangling pointer, followed by access to the memory referenced to by this pointer. While other memory corruption errors, such as buffer overflows [11], are now harder to exploit due to mitigations, few mitigations are deployed in production environments to prevent UAF vulnerabilities, increasing the attack surface significantly [27]. This is why it is crucial to identify and remediate UAF vulnerabilities, as attackers can exploit them to execute arbitrary code or cause a denial of service [38].

Since the source code may not always be available and performance-critical applications and firmware binaries are still written in languages that compile down to binary code, binary analysis remains a crucial area of focus in computer security [34]. Moreover, analyzing a program's source code to verify its properties may not be reliable, as the compiled code may not accurately reflect the original source, which could lead to an attacker inserting malicious code into any compiled program, as noted by [35]. Compiler and toolchain bugs could introduce new vulnerabilities or alter the code's behavior. In such cases, binary analysis becomes the only way to determine the presence of UAF vulnerabilities in the executed code.

Researchers have devised various techniques to identify UAF vulnerabilities in compiled code, including static analysis, dynamic analysis, and symbolic execution [24]. Each method has its advantages and drawbacks. Static analysis examines an entire program without executing it, while dynamic analysis provides insights into a program's behavior during runtime [18]. Symbolic execution can explore all possible paths through a program's code to detect potential UAF vulnerabilities [9]. However, current sound static analysis techniques discussed in [16, 22] suffer from a lack of accuracy, leading to numerous false positives, while symbolic execution faces the path explosion problem. By integrating static analysis with symbolic execution, it would be possible to harness the strengths of both approaches, enhancing the soundness of static analysis and the scalability of symbolic execution. This combination could significantly decrease the number of false positives and assist human analysts in concentrating on actual UAF vulnerabilities, ultimately improving software security overall.

This bachelor thesis aims to develop an automated approach for analyzing UAF vulnerabilities [12] in binary code using symbolic execution to validate the outcomes of static analysis frameworks. The proposed analysis is being built with the `angr` framework, introduced in [34]. The thesis delves into the technical details of creating such an analysis and addressing the path explosion problem. To assess the effectiveness of the proposed UAF vulnerability analysis technique, the Juliet Test Suite [25] is used to test the analysis technique against a range of synthetic test cases that cover a wide range of possible UAF scenarios. Moreover, the analysis's practical performance is evaluated by utilizing real-world binaries.

This thesis seeks to answer the following questions:

- Can the suggested approach accurately detect UAF vulnerabilities?
- Is the analysis fully automated, or does it require human intervention?
- Can Directed Symbolic Execution combined with a static analysis framework effectively address the issue of path explosion?
- Is the approach suitable for analyzing real-world binaries and verifying static analysis tool results?

The thesis is organized as follows: Chapter 2 explores the fundamental concepts and context surrounding UAF vulnerabilities, symbolic execution, the angr framework, and the problem statement. In Chapter 3, the methodology is thoroughly explained, including the UAF analysis functions and the introduction of various heuristics designed to prevent path explosion. These heuristics encompass Under-Constrained Symbolic Execution, Veritesting, and Directed Symbolic Execution. Additionally, the chapter presents the experimental setup for evaluating the proposed method. Chapter 4 details the evaluation results, which involves testing the technique on the Juliet Test Suite and real-world binary files. It demonstrates the effectiveness and performance of the methodology in identifying and addressing UAF vulnerabilities and provides a comprehensive analysis of the proposed strategy's limitations. In Chapter 5, a survey of existing work related to the presented approach is conducted. Lastly, Chapter 6 concludes the thesis by summarizing the key findings made throughout the research while presenting potential ideas for future improvements.

2 Background and Context

This chapter provides a background on UAF vulnerabilities and binary analysis, followed by a discussion on symbolic execution and the `angr` framework. It concludes by presenting the problem statement addressed throughout the thesis.

2.1 Use-After-Free Vulnerabilities

A Use-After-Free (UAF) vulnerability is a memory corruption issue classified under the Common Weakness Enumeration (CWE) code CWE-416 [12]. This vulnerability arises when a program continues accessing an object that has been freed, using a dangling pointer that still references the previously released object [1].

The accessed freed memory in a UAF vulnerability can be located on the heap or the stack. Stack-based UAF vulnerabilities are less common in real-world programs, as static checks can proactively eliminate them. Additionally, due to the frequent allocation and release of stack memory, it becomes challenging for an attacker to reallocate the freed stack memory. Consequently, heap-based UAF vulnerabilities are more commonly exploited and harder to defend against than stack-based UAF vulnerabilities. As a result, this thesis focuses exclusively on finding heap-based UAFs. [27]

A deep understanding of the program is necessary to identify patterns of UAF vulnerabilities. This involves detecting two events:

1. The creation of dangling pointers: A live pointer becomes a dangling pointer when the memory object it points to is freed.
2. The dereference of dangling pointers: A dangling pointer is later used to access previously freed memory, performing read or write operations.

These events can be scattered throughout the code, making it challenging for dynamic and symbolic execution to identify UAFs effectively [24]. Moreover, detecting heap-based UAFs involves identifying potential aliases, which is often difficult to accomplish through static analysis [26], resulting in many false positives.

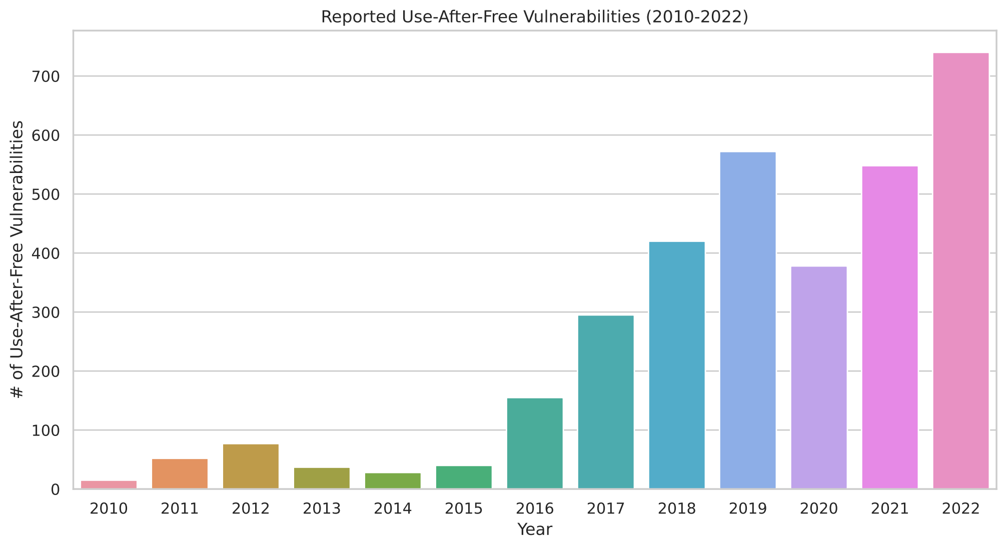


Figure 1: *Number of Use-After-Free vulnerabilities recorded in the National Vulnerability Database (NVD) since 2010 [31].*

However, it is crucial to identify and address UAF vulnerabilities due to their potentially severe consequences, ranging from data corruption to arbitrary code execution [38]. When left unaddressed, attackers can exploit these vulnerabilities to gain unauthorized access to sensitive data, take control of the affected system, or even cause a system crash. The National Vulnerability Database (NVD) has observed a growing trend in UAF vulnerabilities since 2010 [31], as illustrated in Figure 1. This highlights the importance of promptly detecting and remediating UAF vulnerabilities to prevent exploitation by malicious actors.

2.2 Binary Analysis: Static vs. Dynamic

Binary analysis, as explained in [2], is a field dedicated to investigating binary computer programs and their properties, such as machine code and data, to determine their actual characteristics. One of the applications of binary analysis is to detect UAF vulnerabilities. Detection approaches are designed to identify and help developers fix bugs in their programs, regardless of whether there is an attacker, or the UAFs can be exploited. Binary analysis can be classified into two categories: Static analysis and dynamic analysis, both of which can be employed to find UAF vulnerabilities.

Static analysis techniques enable the examination of a binary without executing it, which offers several benefits, such as the ability to analyze the complete binary at once and not requiring a compatible CPU to run the binary. However, a drawback of this approach is that static analysis lacks awareness of the binary's runtime state, making the analysis more challenging. Moreover, while static analysis can confidently determine the absence of vulnerabilities, it frequently generates false positives when identifying the presence of vulnerabilities. Therefore, the detection results often require manual verification. [2, 16, 18, 22]

Dynamic analysis techniques examine a binary's execution in an actual or emulated environment, analyzing it as it executes. Compared to static analysis, this approach is usually more straightforward as it provides complete knowledge of the entire runtime state, including the values of variables and the outcomes of conditional branches. While dynamic analysis does not generate as many false positives as static analysis, it may overlook critical parts of the program since it only observes executed code. [2, 18]

2.3 Symbolic Execution

Symbolic execution, another program analysis technique referenced in [2, 6, 9, 33], utilizes logical formulas to represent program states and verify whether a program violates specific properties. For example, it can ensure that no freed memory is ever read or written. Concrete execution involves running a program with a specific input and exploring a single control-flow path. Consequently, it often only approximates the analysis of the property of interest. Symbolic execution, on the other hand, can simultaneously analyze multiple paths that a program may take under different inputs. The central concept behind symbolic execution is executing a program with symbolic input values instead of concrete ones. As a result, symbolic execution enables sound analyses that can provide strong guarantees on the checked property.

A symbolic execution engine carries out the execution by computing logical formulas over the symbols. These formulas represent the operations performed on the symbols during execution and describe the limits that determine the range of values that the symbols can represent. The symbolic execution engine maintains, for each explored control flow path:

- *path constraints* π that describe the conditions satisfied by the taken branches along the path, and

- a *symbolic memory store* σ that maps registers and memory location to either symbolic expressions or values.

Branch execution updates the path constraints, while register and memory operations update the symbolic store. The combination of the path constraints and all symbolic expressions and mappings is called the *symbolic state*. A Satisfiability Modulo Theories (SMT) solver is utilized to verify if there are any breaches of the property along each explored path and to confirm if the path itself is realizable.

2.3.1 Symbolically Executing an Example Program

When examining the C code in Listing 2.1, the objective is to verify if there are inputs that can cause the UAF at line 11 of the function `example_func`. Since each 4-byte input parameter can have up to 2^{32} distinct integer values, it is unlikely that running the function `example_func` with randomly generated inputs will precisely identify the inputs that trigger the UAF. Symbolic execution solves this limitation by utilizing symbols for inputs rather than concrete values. This technique enables reasoning on classes of inputs instead of single input values, thereby overcoming the challenge of precisely identifying the UAF-triggering inputs.

```c
1  void example_func(int a, int b) {
2      int* ptr = (int*) malloc(sizeof(int)); // allocation
3      int x = 1, y = 0;
4      if (a != 0) {
5          y = 3 + x;
6          if (b == 0)
7              x = 2 * (a + b);
8      }
9      free(ptr);  // deallocation
10     if (x - y == 0)
11         *ptr = 42; // use-after-free
12 }
```

Listing 2.1: *Motivating example of a Use-After-Free vulnerability, inspired by [6].*

Symbolic execution of the function `example_func`, which is shown in Figure 2, can be effectively represented as a tree. Initially, the inputs a and b are symbolic, and the path constraints are *true*. The local variables x and y are initialized, and the symbolic store is updated. The branch at line 3 forks the execution and leads to different statements and assumptions for symbol a. Depending on the branch taken, y is assigned a different value. Upon expanding all execution states until the conditional branch at line 10, input values triggering the UAF for a and b can be determined. In this example, an SMT solver can identify unsafe input parameters by solving the path constraints, yielding $a = 2$ and $b = 0$.

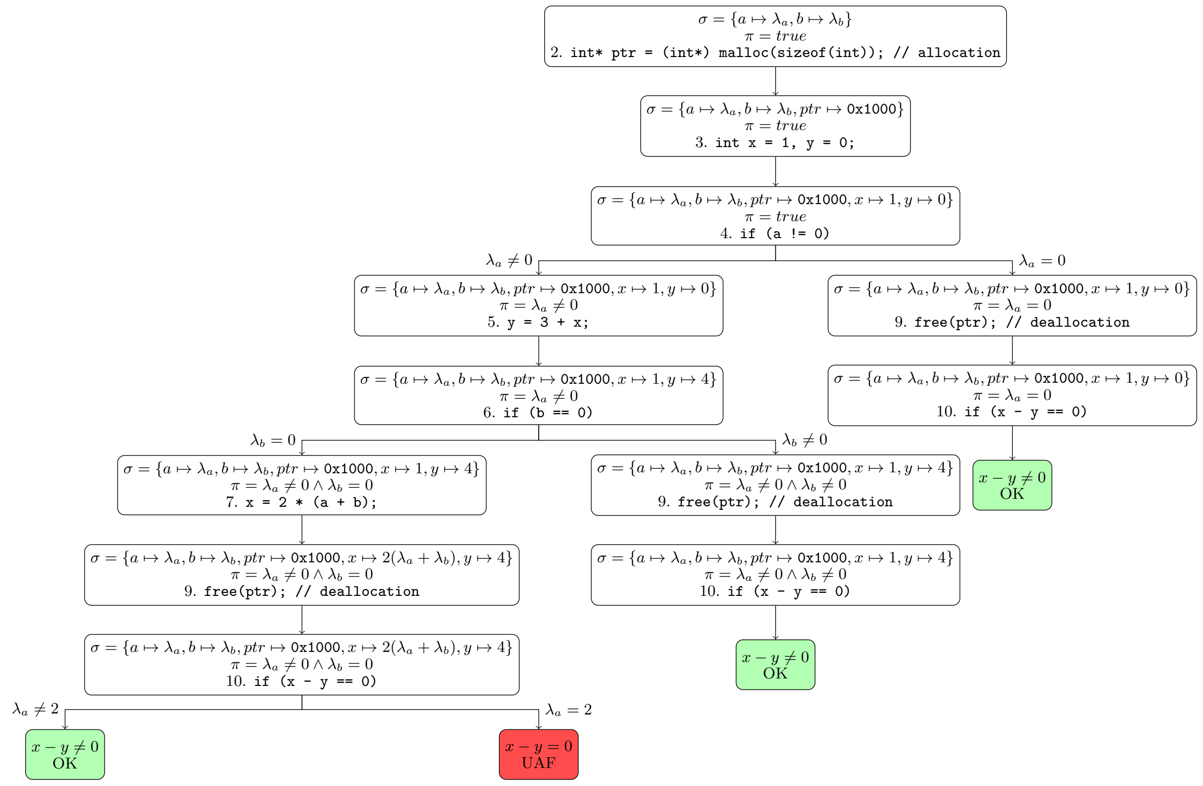

$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b\}$
$\pi = true$
2. int* ptr = (int*) malloc(sizeof(int)); // allocation
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}\}$
$\pi = true$
3. int x = 1, y = 0;
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}, x \mapsto 1, y \mapsto 0\}$
$\pi = true$
4. if (a != 0)
$\lambda_a \neq 0$
$\lambda_a = 0$
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}, x \mapsto 1, y \mapsto 0\}$
$\pi = \lambda_a \neq 0$
5. y = 3 + x;
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}, x \mapsto 1, y \mapsto 0\}$
$\pi = \lambda_a = 0$
9. free(ptr); // deallocation
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}, x \mapsto 1, y \mapsto 4\}$
$\pi = \lambda_a \neq 0$
6. if (b == 0)
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}, x \mapsto 1, y \mapsto 0\}$
$\pi = \lambda_a = 0$
10. if (x - y == 0)
$\lambda_b = 0$
$\lambda_b \neq 0$
$x - y \neq 0$
OK
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}, x \mapsto 1, y \mapsto 4\}$
$\pi = \lambda_a \neq 0 \wedge \lambda_b = 0$
7. x = 2 * (a + b);
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}, x \mapsto 1, y \mapsto 4\}$
$\pi = \lambda_a \neq 0 \wedge \lambda_b \neq 0$
9. free(ptr); // deallocation
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}, x \mapsto 2(\lambda_a + \lambda_b), y \mapsto 4\}$
$\pi = \lambda_a \neq 0 \wedge \lambda_b = 0$
9. free(ptr); // deallocation
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}, x \mapsto 1, y \mapsto 4\}$
$\pi = \lambda_a \neq 0 \wedge \lambda_b \neq 0$
10. if (x - y == 0)
$\sigma = \{a \mapsto \lambda_a, b \mapsto \lambda_b, ptr \mapsto \texttt{0x1000}, x \mapsto 2(\lambda_a + \lambda_b), y \mapsto 4\}$
$\pi = \lambda_a \neq 0 \wedge \lambda_b = 0$
10. if (x - y == 0)
$x - y \neq 0$
OK
$\lambda_a \neq 2$
$\lambda_a = 2$
$x - y \neq 0$
OK
$x - y = 0$
UAF

Figure 2: *Symbolic execution tree of function* `example_func` *given in Listing 2.1, based on [6]. Each execution state shows the statement to be executed, the symbolic store σ, and the path constraints π. Symbolic values are denoted by λ.*

2.3.2 Variants of Symbolic Execution

As detailed in [2, 6], symbolic execution exists in several variations, each with its own set of performance trade-offs, limitations, and degrees of completeness. Many symbolic execution engines implement aspects from multiple symbolic execution variants, allowing users to choose between them.

Traditionally, symbolic execution is a static analysis technique that emulates part of a program and propagates symbolic states with each emulated instruction. This type of symbolic execution is also known as static symbolic execution (SSE). It analyzes all possible paths or uses heuristics to decide which paths to traverse. However, SSE is only sometimes scalable for large programs and has difficulties accurately modeling external interactions like system calls or library calls. Dynamic symbolic execution (DSE), or concolic execution, addresses these limitations by maintaining symbolic states as metadata while using concrete states to drive the execution. This approach explores different paths by flipping path constraints. However, the code coverage depends on the initial concrete inputs and may take time to reach interesting paths. [2]

Symbolic execution engines that explore multiple paths in parallel are called *online*, while engines that explore only one path at a time are called *offline*. Online execution is more efficient as it avoids repeated instruction execution but requires more memory to track all paths simultaneously. In contrast, offline execution may analyze the same code multiple times, having to run the entire program from the start for every program path. However, it does not have the memory cost of online execution. [2]

Additionally, symbolic memory access can be represented fully or partially symbolically, with solutions ranging from modeling all outcomes to concretizing unbounded addresses. Many symbolic execution engines use a combination of these solutions to balance accuracy and efficiency. In conclusion, symbolic execution offers various parameters that can be adjusted to balance the analysis's performance and limitations. [6]

2.3.3 Increasing the Scalability of Symbolic Execution

Theoretically, exhaustive symbolic execution, which involves exploring all possible paths through a program's execution, is a sound and complete approach for any decidable analysis [6]. A *sound* analysis ensures that all reported results are true and accurate, i.e., it produces no false positives. On the other hand, a *complete* analysis ensures that all actual results are reported, i.e., it produces no false negatives. However, the exhaustive symbolic execution approach is not scalable for larger software applications due to the exponential increase in possible paths with the number of program branches. This is commonly referred to as the *path explosion* problem and may result in an infinite number of paths for programs with loops or recursive calls if the termination condition is symbolic. Therefore, there is often a trade-off between the depth and extent of the analysis performed and the resources available in real-world scenarios. As a result, compromises are necessary to the extent of the analysis. For instance, sacrificing soundness or completeness to improve the scalability of the analysis. [6]

Several strategies can be employed to address the challenge of path explosion. One method is to limit the number of iterations for symbolic loops, which can prevent infinite loops and reduce the exploration space [6]. Another approach is to use Under-Constrained Symbolic Execution (UCSE) [17, 32] to analyze individual functions instead of the entire program. In UCSE, symbolic values can be marked as under-constrained, indicating that they may violate preconditions, such as missing constraints on their symbolic values. The downside of UCSE is that it may miss certain program behaviors and increase the risk of reporting false positives. Another way to mitigate the path explosion problem is to incorporate static analysis findings, which can help to prune uninteresting paths and direct symbolic execution more effectively toward areas of code that are more likely to contain vulnerabilities [29]. Lastly, Veritesting [5], a technique that combines static and dynamic symbolic execution, can be used to merge similar states and reduce the number of paths that need to be explored, thereby improving the efficiency and effectiveness of symbolic execution.

The symbolic execution engine `angr`, used in the UAF analysis, provides a framework to address the challenges associated with path explosion. The `angr` framework does not automatically employ these strategies. Instead, this power is given to the user of the framework.

2.4 ANGR

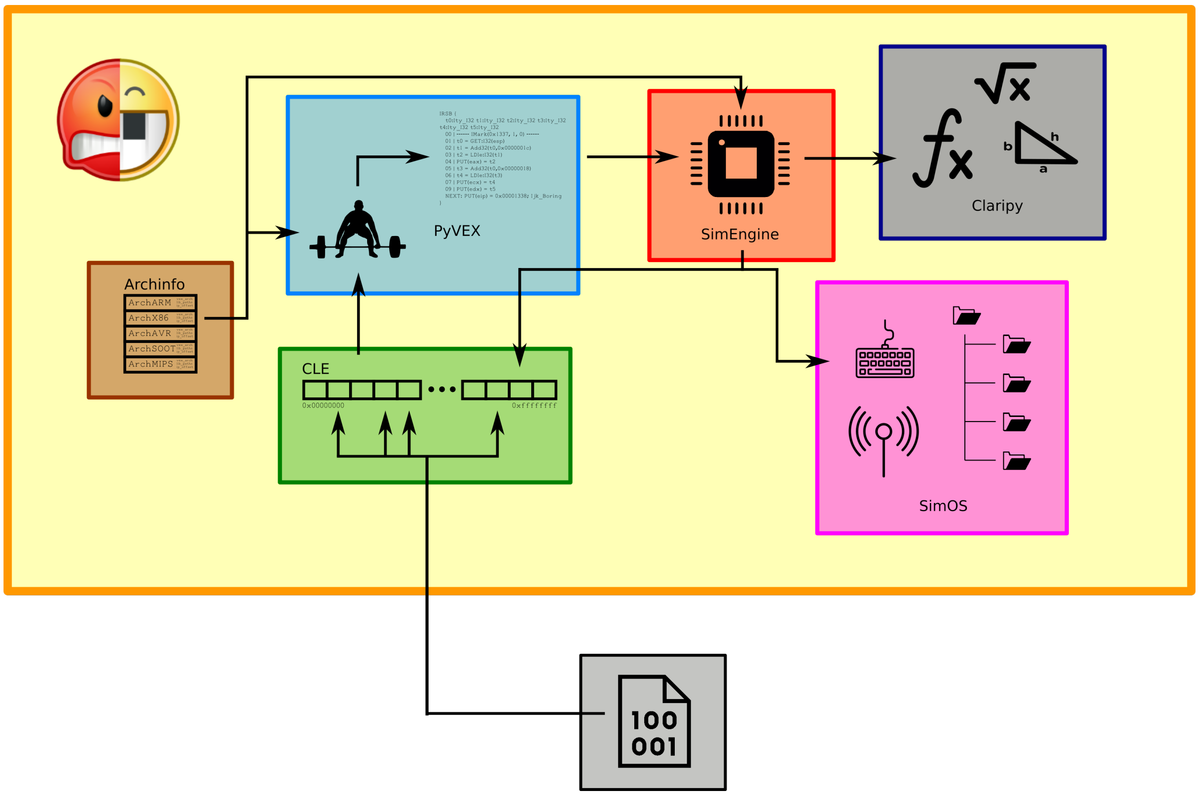
The `angr` framework [3, 34] is an open-source multi-architecture binary analysis platform for Python, with the capability to perform both static and dynamic symbolic analysis. It is developed by the Computer Security Lab at UC Santa Barbara, SEFCOM at Arizona State University, and their associated CTF team, Shellphish [21].

2.4.1 Core Components

As described in [34, 36], to perform programmatic analysis of a binary, the `angr` framework relies on several components that work together to overcome the challenges associated with loading the binary, translating it into an intermediate representation (IR), and exploring the program's state space symbolically. Figure 3 illustrates the main components of `angr`, which include:

- `CLE`, a binary loader;
- `ArchInfo`, an architecture database that provides information on the architecture of the target binary;

- `SimEngine`, an execution engine that simulates program execution and tracks the program states;
- `Claripy`, a data backend that abstracts the differences between static and symbolic domains;
- and `SimOS`, an operating system emulation layer.



```
IRSB {
  t0:Ity_I32 t1:Ity_I32 t2:Ity_I32 t3:Ity_I32
t4:Ity_I32 t5:Ity_I32
  00 | ------ IMark(0x1337, 1, 0) ------
  01 | t0 = GET:I32(esp)
  02 | t1 = Add32(t0,0x0000001c)
  03 | t2 = LDle:I32(t1)
  04 | PUT(eax) = t2
  05 | t3 = Add32(t0,0x00000018)
  06 | t4 = LDle:I32(t3)
  07 | PUT(ecx) = t4
  09 | PUT(edx) = t5
  NEXT: PUT(eip) = 0x00001338; Ijk_Boring
}
```

Figure 3: *Architecture diagram of* `angr` *from [36]. The figure shows the main components of the system. Arrows indicate data flow and dependencies between the components.*

The binary loader named `CLE`, which stands for “CLE Loads Everything”, extracts executable code and data from input formats and creates a memory map of the program. It supports formats like ELF and PE and can load pure binary blobs. `ArchInfo` determines the program’s architecture during the loading process of `CLE` by analyzing its header or using a simple heuristic. Once the code is loaded, `angr` searches for an execution engine to interpret it. `angr`’s default engine uses an IR called VEX to support multiple architectures, including x86, MIPS, ARM, and PowerPC. The execution engine processes a block of instructions, generates possible program states, and collects constraints to capture the conditions for each branch. Instead of operating on concrete values, engines may work with complex symbolic expressions. `Claripy` creates and combines these expressions, solving them using Microsoft's Z3 [15] SMT solver. To handle program states involving files, network, and operating system functionality, `angr` uses `SimOS`, a higher-level abstraction. A `SimOS`, such as `SimLinux`, is created based on `CLE`'s guess, defining OS-specific features, system calls, and symbolic summaries of standard library functions called `SimProcedures`.

These components create a symbolic execution environment to explore efficiently and reason about programs using symbolic values for path exploration and vulnerability identification.

2.4.2 Simulated Program States

In `angr`, a `SimState` is an object that represents the state of a simulated program at a certain point in its execution. It is the primary object used when performing symbolic execution and analysis of a program. During symbolic execution, a `SimState` can be progressed through stepping, which produces a `SimSuccessors` object. Because of symbolic bit-vectors and conditional branches in the code, symbolic execution can generate multiple successor states. A `SimState` can be customized to specific requirements through state options, which are maintained in a set and can be appended or deleted when creating a new state. In addition to options, `SimState` objects have plugins that store and manage the state's data. Examples of state plugins include memory, registers, and the solver. [3, 34]

2.4.3 Simulation Manager

The `SimulationManager` serves as the primary control interface for enabling the symbolic engine to carry out symbolic execution across multiple states simultaneously. This is achieved using search strategies to navigate a program's state space. The states are organized into stashes, which can be advanced, filtered, merged, and moved. The `SimulationManager`'s fundamental ability is to progress all states in a stash by one basic block. When a state encounters a symbolic branch condition, both of its successor states are included in the stash and can be advanced together. Additionally, `angr` features several exploration techniques that allow customization of the `SimulationManager`'s behavior. While the default strategy is a breadth-first search (BFS), implementing other exploration techniques can result in different search strategies, such as depth-first search (DFS). [3, 34]

2.4.4 Analyses

`angr` aims to simplify analyzing binary programs by providing building blocks for various analyses, including static and dynamic techniques. These building blocks can be combined to leverage their different strengths. The `Project` object, which represents a binary and its associated libraries, is the entry point into these analyses. From the `Project`, all functionality of `angr`'s submodules can be accessed, including creating states, retrieving intermediate representations of basic blocks, and hooking binary code with Python functions. When `angr` discovers information about a binary, it stores it in the corresponding `Project`'s knowledge base. This shared knowledge base allows analyses to collaborate and discover more information about the application. [3, 34]

2.5 Problem Statement

This thesis aims to use `angr`'s building blocks to create an analysis that can find UAF vulnerabilities in binaries using symbolic execution. The analysis allows users to choose between different exploration strategies, such as DFS, LoopSeer, UCSE, Veritesting, and Directed Symbolic Execution, depending on their specific use case. This approach intends to verify the results of static analysis tools and improve the accuracy of the UAF vulnerability detection process. The primary objective is to provide an efficient and effective UAF analysis for binary programs while discovering the best combination of techniques to tackle the path explosion problem.

3 Methodology

In order to detect UAF vulnerabilities, symbolic execution can be used to explore all possible paths through a program and identify any points where a memory region might be accessed after it has been freed. This can be done by tracking the state of memory regions as the program executes and checking whether memory accesses violate the expected safety properties.

However, symbolic execution is susceptible to path explosion, where the number of possible program paths increases exponentially, leading to an impractically large number of paths to analyze. Various techniques have been developed to overcome this challenge, including bounding symbolic loops, UCSE, Veritesting, and Directed Symbolic Execution using the results of static analysis tools to prune paths. By combining these techniques, symbolic execution can be a powerful approach for detecting UAF vulnerabilities while effectively managing the potential path explosion during analysis.

3.1 Use-After-Free Analysis

The symbolic execution-based analysis uses `angr` version 9.2.48 to detect potential UAF vulnerabilities in binaries. The `SimState` and `SimulationManager` objects provided by `angr` are used to manage program states and the analysis process. Additionally, a `SimState` option is used to symbolically initialize uninitialized memory locations in the program, allowing for more accurate modeling of program behavior. By hooking memory allocation and deallocation functions and setting breakpoints on every memory read and write operation, the analysis identifies program states that may contain a UAF. Manual inspection of these states is required to confirm the presence of a security vulnerability.

3.1.1 Hooking Dynamic Memory Functions

The analysis employs function hooking to intercept dynamic memory allocation and deallocation functions, such as `malloc` and `free`. This is achieved by replacing the function calls with custom Python code that records the pointer, size, and free status for allocated memory. When an allocation occurs, the analysis allocates memory, saving the returned pointer and allocator size, and marks the region as not free. When deallocation occurs, the analysis records the free site and labels the region as free.

However, the memory is not truly deallocated, guaranteeing that each pointer returned is unique. This is crucial because if an allocator function were to return a pointer that had already been freed but with a different size, it would complicate analysis as it would be uncertain whether memory access occurs in a freed region. The default memory size is increased to 1MB to mitigate the risk of exhausting the heap space.

Recording deallocations still works if no allocation function has previously been called by using a symbolic address instead. This may occur when using `scanf` with the `%ms` specifier, where memory is dynamically allocated for the input string. To ensure that the allocation and deallocation data is saved for each state and can be used for state merging, a `SimPlugin` is created. This plugin uses deep copies to maintain state independence, which is essential for preventing false detections during analysis.

In the `angr` framework, there are two available heap implementations: `SimHeapBrk` and `SimHeapPtMalloc`. `SimHeapBrk` is a basic heap implementation based on the Unix `brk` system call. It stores minimal metadata and is set as the default option. Meanwhile, `SimHeapPtMalloc` is a free list heap implementation derived from `dlmalloc`, which is utilized in glibc. This implementation is more sophisticated than `SimHeapBrk`, aiming to provide more accurate heap modeling. In practice, `SimHeapPtMalloc` tends to perform significantly slower without any notable improvements. As a result, the analysis utilizes `SimHeapBrk`.

3.1.2 Vulnerability Detection

The method of detecting UAF vulnerabilities involves monitoring memory access operations during program emulation. This is achieved by setting breakpoints for every memory read and write operation and checking whether the operand falls within a previously freed heap buffer. The algorithm depicted in Figure 4 verifies whether a memory object lies within the bounds of a previously released heap buffer's memory address and its corresponding size by examining the recorded allocation sites each time an object is accessed. If the memory object is found to be allocated, the access is considered valid, but if it has been deallocated, the program flags it as a UAF vulnerability.

```
Require: state, use-addr
foreach memory block in state's heap do
    if memory block is marked as free then
        in-range ← (block-start ≤ use-addr) ∧ (use-addr < block-end);
        if the size of the memory block is symbolic then
            set min and max constraints on the size;
        end
        if use-addr is symbolic and not uninitialized then
            set heap bounds constraints on the use-addr;
        end
        if use-addr is uninitialized or block-start is uninitialized then
            use-after-free ← (use-addr == block-start);
        else
            use-after-free ← state's solver is able to satisfy in-range;
        end
        if use-after-free then
            save a copy of the state;
        end
    end
end
```

Figure 4: *Algorithm for checking a memory access for a Use-After-Free vulnerability.*

As the program is executed symbolically, the use-address, freed-pointer, and size can also be symbolic. Therefore, `Claripy` is used to check if there is a satisfiable assignment where the operand is within a previously freed buffer. When a satisfiable assignment is found, the analysis flags it as a UAF vulnerability and saves a copy of the current `SimState` in a list.

The analysis imposes certain constraints to minimize false positives. These include limiting the allocation requested size to the maximum variable size and ensuring that the use address falls within the heap base and heap end range. The decision to set the maximum variable size is due to the limitations of `SimHeapBrk`, which is responsible for automatically concretizing symbolic sizes to this maximum value.

Moreover, the analysis addresses the issue of uninitialized symbolic values, which can assume any value without constraints. This is accomplished by verifying if the use and free address symbols are identical. This step is crucial because the solver would always return a satisfiable assignment without it, given that the unconstrained symbol can take any value.

This approach is more favorable than concretizing the symbolic values, considering the infinite possibilities and the lack of context.

Output Format

A specific output format is designed to verify that the identified UAF states represent genuine vulnerabilities. This format entails selecting only the unique address states with the lowest depth from the list of UAF states. This helps eliminate duplicates and allows users to concentrate on distinct UAF states. The generated output, seen in Listing 3.1, comprises all the essential information required to identify a potential UAF state. This encompasses the state address, call stack, allocation site, free-site, use-site, free-ptr, use-ptr, size, stdin, general-purpose register values, and path. The data provided in the output can subsequently be employed to prioritize and address potential security risks.

```
[+] Use-After-Free: <SimState @ 0x401189> in main may access a dangling pointer
Call stack: __libc_start_main -> main
Allocation-site: @ 0x401159
Free-site:       @ 0x40116b
Use-site:        @ 0x401189
Free-ptr:        @ <BV64 0xc0000f40>
Use-ptr:         @ <BV64 0xc0000f40>
Size:              <BV64 0x4>
Registers:
    rax: <BV64 0xc0000f40>
    rcx: <BV64 0x0>
    rdx: <BV64 0x7fffffffffff0010 + 0x8 * mem_7fffffffffff000...
    rbx: <BV64 reg_28_8_64{UNINITIALIZED}>
    (truncated...)
Path:
    1: 0x401060 - _start
    2: 0x500000 - __libc_start_main
    (truncated...)
    10: 0x401189 - main
```

Listing 3.1: *Script output indicating the detection of a Use-After-Free vulnerability.*

3.2 Dealing with Path Explosion

The analysis employs the `angr` exploration strategies DFS, LoopSeer, Veritesting, and the custom-implemented techniques UCSE and Directed Symbolic Execution to address the challenge of path explosion. DFS reduces memory consumption, while LoopSeer limits the exploration of symbolic loops. UCSE focuses only on selected functions while Veritesting merges states to reduce the number of paths. Directed Symbolic Execution leverages the results of static analysis tools to prune uninteresting paths. By combining these techniques in various ways, the analysis has the potential to mitigate path explosion, enabling the detection of UAF vulnerabilities in larger and more complex programs. However, each technique has strengths and limitations, and their effectiveness may vary depending on the program being analyzed. The analysis allows users to choose which heuristics they wish to use to address the challenge of path explosion.

3.2.1 Reducing Memory Consumption

During analysis, path explosion can result in memory exhaustion and crashes as many states accumulate for subsequent steps. `angr`'s default exploration strategy, breadth-first search (BFS), processes all states simultaneously, rapidly increasing the number of states. In contrast, depth-first search (DFS) exploration is an alternative that maintains only one active state at a time and defers others until a dead end or error is encountered. In this context, BFS corresponds to online symbolic execution, while DFS represents offline symbolic execution. Offline symbolic execution helps reduce state maintenance but may lead to longer analysis times, as paths are examined individually instead of in parallel.

An additional exploration technique is implemented to save memory that automatically discards irrelevant states unsatisfiable or discarded by other techniques. The MemoryWatcher exploration technique monitors system memory availability between `SimulationManager` steps. If memory reaches critically low levels, exploration halts to prevent crashes. While limiting memory consumption makes the analysis feasible, it does not resolve the path explosion problem. This trade-off merely exchanges memory for running time, potentially leading to an analysis that never terminates.

3.2.2 Simulating Library Calls

A viable strategy to address path explosion involves bypassing the loading of external libraries and replicating their effects through simulation. This can be accomplished using `SimProcedures`. In angr, `SimProcedures` are Python functions that imitate the effects of library functions on the program state. Utilizing `SimProcedures` helps preserve the integrity of symbolic execution, as they mimic the behavior of library functions without requiring symbolic execution of the library code. This method significantly reduces the number of states generated during analysis and dramatically improves the speed of the Control Flow Graph (CFG) recovery process.

However, not all library functions may have corresponding `SimProcedures`, leading to the use of a generic stub `SimProcedure` called `ReturnUnconstrained`. This can result in inaccuracies or loss of information during analysis. Additionally, complex library functions might not be fully covered by `SimProcedures`, causing incomplete analysis. Despite these drawbacks, using `SimProcedures` remains a practical option for handling library calls in symbolic execution, as it generally offers a more manageable analysis while mitigating path explosion.

3.2.3 Bounding Loops with LoopSeer

Loops are a significant cause of path explosion in symbolic execution. When executing loops, each iteration creates a conditional branch in the execution tree, similar to an if-goto statement. If the loop condition involves one or more symbolic values, the number of generated branches may be infinite. Therefore, it is crucial to restrict symbolic loops effectively to avoid path explosion. A common approach is to bind the loop exploration of symbolic loops up to a limited number of iterations.

The analysis leverages `angr`'s LoopSeer exploration technique to efficiently discard states that appear to be trapped in a loop. LoopSeer utilizes a normalized CFG to track loops and their trip counts. Normalizing a CFG involves simplifying and standardizing it for more accessible analysis by eliminating unnecessary nodes and streamlining the structure. The normalized CFG is also used to perform a loop finder analysis, identifying all loops present in the binary. Looping states are stored in a *spinning* stash. The analysis configures LoopSeer to avoid limiting iterations for loops with only one successor, considered concrete loops and sets the bound for symbolic loops to one. By default, LoopSeer limits iterations for all loops, including concrete loops, which may result in overlooking interesting paths. While not limiting concrete loops may capture more paths, it can still lead to path explosion and impact the analysis's efficiency by generating a large number of paths to explore.

3.2.4 Merging States with Veritesting

Veritesting [5] is a powerful technique that addresses the problem of state explosion in loops. The key idea behind Veritesting is to merge multiple program paths into a single path using a disjunctive normal form (DNF) representation of the program state. This merging process is achieved through a combination of static and dynamic symbolic execution. Static symbolic execution identifies the areas that can be merged, while dynamic symbolic execution explores the state space of the merged region. By merging the paths, Veritesting reduces the number of branches that need to be explored, thereby mitigating the path explosion problem.

`angr` utilizes Veritesting through two techniques: an exploration technique and an analysis. The exploration technique is used to update the main `SimulationManager` with the merged states, while the analysis technique is used to identify regions of code that can be efficiently merged. The analysis begins by creating an initial region and adding the given state to it. Then the region is explored by following control flow edges and collecting states. As the exploration continues, states within the region are merged whenever possible, creating a new region with merged states. The process of exploring and merging continues until a fixed point is reached or a specified maximum number of iterations is exceeded. The Veritesting analysis results in a final state manager that contains the merged states and the associated control flow information. This final manager is then used in the Veritesting exploration technique to update the main `SimulationManager` with the merged states.

It is crucial to recognize that Veritesting, despite its advantages, is not a universal solution and may only be appropriate for some programs. For example, the efficient path merging employed by Veritesting can introduce complex expressions that may overload the constraint solver, potentially making some vulnerabilities unreachable within an acceptable timeframe, given that solving constraints is an NP-complete problem. While Veritesting is generally effective at detecting shallow bugs that symbolic execution may miss due to path explosion, its utility for programs with intricate control flow or non-deterministic behavior may be more limited.

3.2.5 Targeted Analysis with UCSE

Another way to prevent path explosion is to isolate the code that needs to be analyzed, such as a function, from the rest of the system and examine it separately. Under-Constrained Symbolic Execution (UCSE), as discussed in [17, 32], is a modified form of symbolic execution that achieves this by identifying the function's symbolic inputs and relevant global data as under-constrained. Marking a variable as under-constrained means that the analysis does not consider any constraints on its value that would have been collected along the path from the program's starting point to the function.

UCSE is implemented by starting the analysis for every function or specific defined function addresses, but it only targets functions that are not `SimProcedures` or external functions. The implementation uses `angr`'s `call_state` constructor to create a state ready to execute a given function and automatically designates the function input as under-constrained. When analyzing all functions, the analysis uses a modified version of the state option `CALLLESS`, which skips function calls by interpreting call instructions as if they store an unconstrained value in the return value register. However, the modified `CALLLESS` does not skip the `SimProcedures` for dynamic-memory functions like `malloc` and `free`, to still detect UAF vulnerabilities. This modified version is used to improve the scalability of the analysis by avoiding analyzing certain parts multiple times. When analyzing specific functions, the `CALLLESS` modification is not used to improve the soundness and completeness of the analysis. Skipping function calls during analysis can result in false negatives in identifying UAF vulnerabilities. This is because precise inter-procedural reasoning is necessary to accurately detect such vulnerabilities, as the allocation, deallocation, and use of the same memory chunk can be widely scattered across different functions in the code.

UCSE may generate false positives by detecting vulnerabilities in individual functions that may never manifest when executed within the larger context of a program. Nevertheless, it remains a powerful tool for detecting UAF vulnerabilities in larger programs. However, to avoid the risk of overlooking genuine issues, inter-procedural reasoning support is imperative to prevent false negatives.

3.2.6 Pruning Paths Using Static Analysis Results

In order to determine whether static analysis vulnerability-finding tools are accurately identifying vulnerabilities, it is necessary to examine only the paths that could lead to the reported addresses. This is known as the *address reachability* problem, which seeks to identify feasible paths to a given program address. Directed Symbolic Execution, introduced by [29], is used to tackle this problem. The approach prioritizes exploring potentially vulnerable paths during symbolic execution while discarding uninteresting paths to avoid the path explosion problem. An analysis is conducted to guide the exploration, by identify all program CFG nodes that involve deallocations and lead to the target address. The exploration method then uses the resulting set of nodes to eliminate any states that do not satisfy this requirement.

To create a list of nodes that are potentially interesting to explore using symbolic execution, a preprocessing pass is used. This involves analyzing the program's CFG to identify paths that could lead to UAF vulnerabilities. Specifically, paths that reach particular goal addresses after encountering memory-free operations are sought. The primary method takes the CFG, a program state, a maximum number of steps, and a list of goal addresses found by a static analysis tool. It uses breadth-first search to traverse the nodes in the graph and identify nodes that represent memory-free operations, such as `free`, `realloc`, or `operator delete`. For each such node, the method retrieves its predecessors and successors and checks if they can reach the goal addresses within the specified number of steps. The resulting nodes are returned as a sorted list of tuples, each containing the start and end address of the node. The exploration technique can then utilize this list to focus on potential UAF vulnerabilities in a program.

The exploration technique for Directed Symbolic Execution requires a sorted list of relevant nodes as input. This technique categorizes states for every execution step into two groups based on whether they can reach the UAF address. A binary search determines if the state address is contained in a node in the list. The states that reach the UAF address are given priority, while those that do not are deprioritized and discarded. Following that, the exploration technique then focuses on the prioritized group and proceeds with symbolic execution.

The approach discussed has significant potential to mitigate the path explosion problem and focus analysis on only the relevant paths, but its effectiveness is contingent on two factors: The quality of results produced by the static analysis tool and mainly the accuracy of `angr`'s CFG recovery process. Since the correct and precise recovery of the CFG is a big challenge of binary-level static analysis [30], it remains questionable whether the CFG produced by `angr` is always correct. Assuming accurate CFG recovery, this approach can effectively tackle the address reachability problem and guide the symbolic execution towards potentially vulnerable paths. Moreover, this approach is compatible with previously mentioned methods to prevent path explosion. By focusing only on the relevant paths, the number of paths generated is significantly reduced, lowering the computational overhead of the analysis. Therefore, despite the limitations of this approach, it can be a valuable tool for identifying UAF vulnerabilities in programs and can be used in conjunction with other techniques to enhance the efficiency and effectiveness of the analysis.

3.3 Setup

The Juliet Test Suite 1.3 for C/C++ [4, 7] is employed to evaluate the accuracy of the UAF analysis, specifically focusing on the test cases for CWE-416 as the ground truth. Furthermore, the practical performance of the analysis is evaluated by examining real-world binaries containing UAF CVEs. This evaluation aims to determine the scalability of the analysis on real-world binaries and compare the various methods for handling path explosion. Both evaluations are conducted on a 64-bit Ubuntu Jammy system equipped with 128 GB of RAM and powered by 16 Intel Xeon E5-2650 v3 CPU cores, each running at 2.30 GHz.

3.3.1 Juliet Test Suite

The Juliet Test Suite, developed and maintained by the National Institute of Standards and Technology (NIST), is a comprehensive collection of small artificial programs containing intentional flaws. It is widely used to assess the effectiveness of vulnerability detection tools. The Juliet 1.3 version was released on October 2017, and its test cases are publicly accessible for download [25].

Test cases consist of buildable code snippets designed to evaluate vulnerability analysis tools. Each test case targets a specific flaw, such as a UAF, in this context. Alongside the flawed construct, every test case includes a non-flawed construct that performs a similar function. A unique identifier for each test case combines the CWE entry, a functional variant name specifying the intentional flaw, and a two-digit flow variant number indicating the data and control flow type. For example, the test case `CWE416_Use_After_Free__malloc_free_char_43` shows how C++ code can be vulnerable when using `malloc` and `free`. Flow variant 43 indicates data flow between functions in the same source file using a C++ reference.

To evaluate the effectiveness of the analysis in identifying UAF vulnerabilities, 393 non-vulnerable test cases labeled as *good* and 393 vulnerable test cases labeled as *bad* under CWE-416 are compiled into binary files. When the analysis tool is run on a bad test case, it should ideally report one UAF flaw in a function with the word "bad" in its name, which is considered a true positive. In structs, the analysis should report exactly two UAFs in bad functions, as the entire struct is freed, but each variable used is considered one UAF. The result is deemed a false negative if the tool fails to report a UAF in a bad function or class within a test case. Conversely, running the analysis on a good test case should not report any UAF flaws in a function with the word "good" in its name. An incorrect UAF report in a good function is considered a false positive. The UAF analysis's accuracy is determined by assessing its true positive and false positive rates.

3.3.2 CVEs in Real-World Binaries

Real-world binaries known to have UAF vulnerabilities and corresponding Common Vulnerabilities and Exposures (CVEs) are used to assess the effectiveness and scalability of the UAF symbolic execution analysis and its ability to verify the outcomes of static analysis tools. The selected binaries for this evaluation comprise:

- `GoHttp` compiled for x86-64 with CVE-2019-12160,
- `readelf` compiled for x86-64 with CVE-2018-20623, and
- `curl` compiled for ARM64 with CVE-2018-16840 and CVE-2020-823.

These binaries are chosen because they represent various types of software and architectures, such as web servers, file-parsing tools, and command-line utilities. Additionally, they cover both x86-64 and ARM64 architectures.

The static analysis tool used in this evaluation is the `cwe_checker` [13]. Its output includes a list of detected vulnerabilities and their locations that require verification, which also serves as the input for the Directed Symbolic Execution. The `cwe_checker` annotates the addresses of dangling pointer accesses, and function calls with dangling pointers in their parameters. To ensure that the Directed Symbolic Execution executes these function calls, the addresses after the calls are used. The symbolic execution analysis results are compared to the static analysis tool results, with discrepancies investigated to determine their causes. If a UAF is present in both sets of results, a manual examination must confirm it as the CVE vulnerability.

Different heuristic approaches for addressing path explosion are compared by analyzing each binary with Directed Symbolic Execution, UCSE, and Veritesting. The comparison considers vulnerability detection, analysis time, memory usage, and the maximum number of active states generated. Directed Symbolic Execution and UCSE are employed with LoopSeer to bound symbolic loops, while Veritesting does not support merging loop data and thus cannot use LoopSeer.

Due to the potential for an infinite number of paths, exploration is restricted by a maximum path length of 3000 and a three-hour timeout. The analysis uses BFS and limits the memory usage to 120GB to reduce the runtime. Using DFS would allow for a lower memory limit but would increase the runtime significantly.

4 Evaluation

In this evaluation, the effectiveness of the UAF symbolic execution analysis is assessed on both synthetic test cases from the Juliet Test Suite and real-world binaries. The primary goal of this evaluation is to determine whether the UAF analysis technique can detect UAF vulnerabilities in various types of programs and assess its scalability in handling real-world applications. Additionally, this evaluation discusses the limitations of the UAF analysis approach, explicitly addressing the challenges posed by path explosion and the reliance on the `angr` framework.

4.1 Juliet Test Suite

The UAF symbolic execution analysis was conducted on all 786 CWE-416 test cases from the Juliet Test Suite, taking 46 minutes and 29 seconds to complete. The results are presented in Table 1, showing a 100% accuracy rate with zero false positives and zero false negatives. This indicates that the technique can detect UAF vulnerabilities in simple, artificial code. The approach presented in the thesis outperformed SATracer [28] and UAFDetector [41] with a slightly better performance. SATracer achieved an accuracy of 99.36% with zero false positives and five false negatives, while UAFDetector achieved an accuracy of 99.62% with zero false positives and three false negatives.

The Juliet Test Suite is not susceptible to the path explosion issue as it is designed with relatively simple test cases. Most of the programs in the test suite have only one path leading to either a "bad" or "good" function and do not require input or input files. This limited number of paths makes it easier for symbolic execution analysis to explore all possible paths without experiencing the path explosion problem, which commonly arises in more complex programs with numerous branches and loops.

Additionally, it should be noted that the exceptional accuracy demonstrated by the UAF symbolic execution analysis on the Juliet Test Suite does not guarantee that the same level of performance will be achieved on real-world binaries. This is because real-world applications tend to be more complex, with intricate code structures and dependencies that cannot be

Table 1: *Summary of Juliet Test Suite results for CWE-416.*

CWE-416: Use-After-Free	
Positive cases:	393
Negative cases:	393
True Positives:	393
False Positives:	0
True Negatives:	393
False Negatives:	0
True Positive Rate:	100%
True Negative Rate:	100%
Accuracy:	100%

captured by synthetic test cases. Moreover, the test cases in the Juliet Test Suite do not cover concurrent programming scenarios, where UAF vulnerabilities can arise due to improper synchronization or race conditions.

Therefore, a second evaluation of actual binaries is necessary to understand better the scalability of analyzing real-world binaries and validate the results of static analysis tools.

4.2 Real-world Applications

The binary files were initially analyzed using the `cwe_checker` static analysis tool. Following this, the UAF analysis was conducted via separate runs for Directed Symbolic Execution, UCSE, and Veritesting. The entire process took 18 hours and 21 minutes to finish. Table 2 presents the results. While the `cwe_checker` identified 216 UAF vulnerabilities, none were reported by the symbolic execution analyses. This suggests that the UAF analysis cannot detect UAF vulnerabilities in real-world binaries. As a result, more than the current form of analysis is needed to verify the findings of static analysis tools.

Table 2: *Comparative analysis of symbolic execution techniques and the `cwe_checker` static analysis tool for detecting UAF vulnerabilities in known CVEs. The dash symbol indicates failed execution.*

Known UAFs		Number of reported UAFs			
Identifier	Program	cwe_checker	Directed	UCSE	Veritesting
CVE-2019-12160	GoHttp	1	0	0	0
CVE-2018-20623	readelf-2.31.1	7	0	0	-
CVE-2018-16840	curl-7.61.1	103	0	-	0
CVE-2020-8231	curl-7.71.1	105	0	-	0
Total		216	0	0	0

Three symbolic execution runs were terminated prematurely by the system: the UCSE runs for CVE-2018-16840 and CVE-2020-8231, and the Veritesting run for CVE-2020-8231. This termination occurred due to the following reasons:

- In the case of the UCSE runs, many active states were created within a single step. The MemoryLimiter only checks the memory limit after each step, so it could not halt the execution in time.
- For the Veritesting run, `angr` implements the analysis using a separate `SimulationManager`. This approach effectively bypasses the timeout and memory limit applied to the original `SimulationManager`, leading to a failure in the run.

The primary challenge that hinders the symbolic execution analysis from detecting UAF vulnerabilities is the path explosion problem, which will be discussed in 4.2.1. Additionally, the limitations of the analysis and its underlying symbolic execution engine contribute to detection failures, which will be covered in 4.3.

4.2.1 Comparison of Path Explosion Handling Methods

Path explosion remains a challenge for all three methods, leading to either memory exhaustion due to the need to maintain many active states or a timeout due to increased computation for the analysis and SMT-solver. A comprehensive comparison is carried out, examining the performance of these methods in terms of runtime, memory consumption, and the maximum number of generated states.

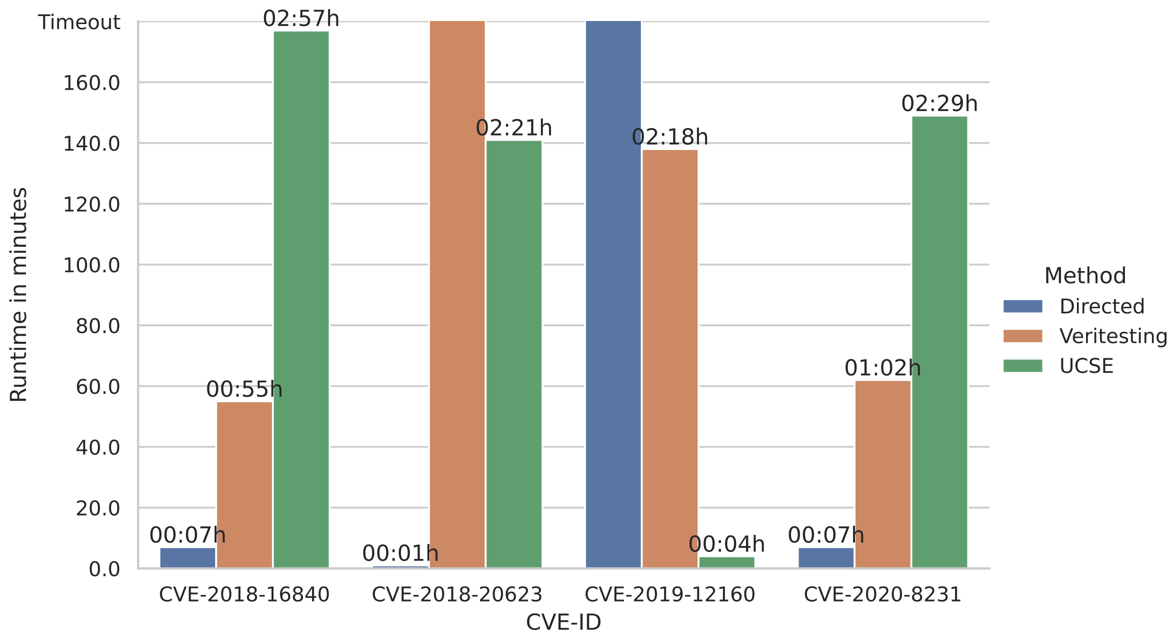


Figure 5: *Runtime for methods by CVE-ID.*

In the runtime comparison shown in Figure 5, Directed Symbolic Execution had the shortest average runtime, followed by Veritesting and UCSE. However, one directed and one Veritesting run timed out. The timeout for the directed run was found to be caused by infinite concrete loops in the `GoHttp` binary. As LoopSeer only bounds symbolic loops, this has the consequence that the symbolic execution also runs indefinitely and never terminates until it times out or runs out of memory. The 1-minute runtime of the directed run for the `readelf` binary was cut short because of unresolved indirect jumps in the recovered CFG, preventing the analysis from reaching the main function. This issue is critical for Directed Symbolic Execution, which relies on precise CFG recovery.

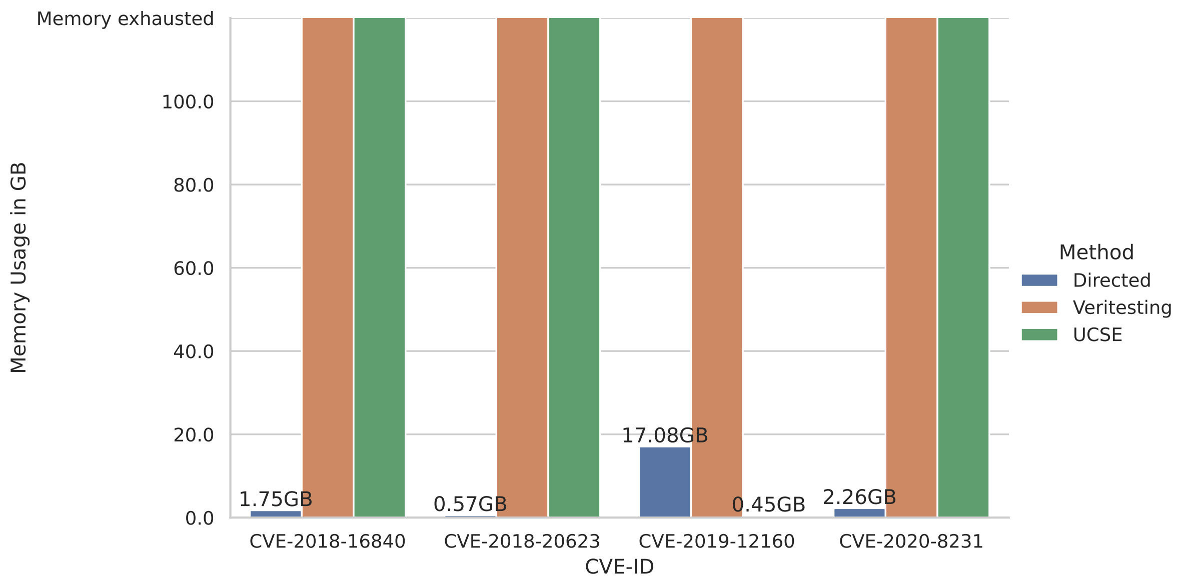


Figure 6: *Memory Usage for methods by CVE-ID.*

Regarding memory usage, Figure 6 shows that `angr`'s Veritesting implementation exceeded the memory limit in every run. This could be attributed to `angr` performing Veritesting at every conditional jump rather than only at symbolic branch starting points, as suggested in the original paper. Additionally, Veritesting creates a new `SimulationManager` for every step, accumulating unused executed states that should be dropped but are not. Addressing these issues would necessitate a custom Veritesting technique. Except for one UCSE run, all other UCSE runs also hit the memory limit. Directed Symbolic Execution timed out once but never reached the memory limit.

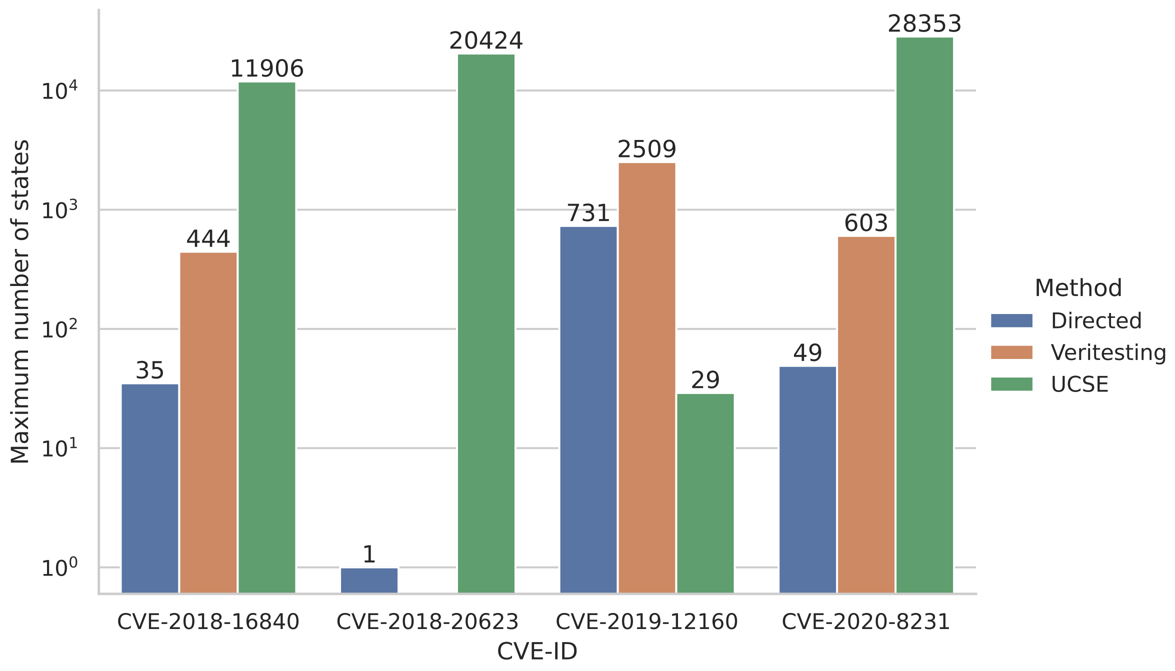


Figure 7: *Maximum number of active states for methods by CVE-ID.*

In Figure 7, it becomes evident that UCSE generates the highest number of active states. This can be attributed to each state saving less information within its history plugin, consequently consuming less memory. UCSE operates by executing a single function at a time without venturing into any nested function calls. However, the presence of nested concrete loops and branches is sufficient for generating a substantial amount of active states. Despite Veritesting's ability to merge similar states, it still produces more active states than Directed Symbolic Execution. In contrast, Directed Symbolic Execution generates the smallest number of active states, underscoring the effectiveness of path pruning using static analysis tools in reducing the number of active states.

The most promising approach for addressing the path explosion problem is the Directed Symbolic Execution method. Nevertheless, there is an opportunity for further enhancement by integrating additional information from static analysis tools, such as identifying the creation sites of dangling pointers and determining which function calls to skip or enter. It would also benefit the Directed Symbolic Execution to use the same CFG as the static analysis tool.

Applying the UCSE method to every function is not scalable. A more efficient approach would be to apply UCSE only to functions that generate dangling pointers and track execution from that point forward. The problem of the infinite loops should also be addressed by improving the LoopSeer to bounding infinite loops to a specific amount of iterations. Although the concept of Veritesting appears promising, its current implementation in `angr` does not effectively mitigate path explosion.

Future work should closely integrate static analysis tools' results with Directed Symbolic Execution and UCSE. Additionally, efforts should be made to develop a modified Veritesting method that is more memory-efficient.

4.3 Limitations

The primary challenge encountered in the analysis is scalability. While it yields satisfactory results when applied to small binaries like those in the Juliet Test Suite, its performance decreases significantly when analyzing larger binaries. This limitation may be acceptable for examining smaller binaries in embedded devices but is less suitable for handling more extensive binaries. In addition to the path explosion problem affecting scalability, the analysis completeness is influenced by the following limitations.

4.3.1 SimProcedure Inaccuracy

To make symbolic execution more tractable, `angr` replaces common library functions with summaries written in Python called `SimProcedures`. While `SimProcedures` mitigate path explosion that would be caused by running library functions, they do not always accurately simulate all library functions [23]. For example, during the evaluation process, several `SimProcedures` were found to be inaccurate:

- The `fgets` `SimProcedure` had to be manually patched to support tracking the newline character, which is essential for covering paths with this condition.
- The `getenv` `SimProcedure` had to be modified because it created symbols for unknown environment variables without constraints but did not label them as uninitialized, leading to the generation of false positives in the curl binaries.
- The simulation of `pthreads` halted `angr`'s execution engine, suggesting that the analysis cannot cover programs using threading.

These examples demonstrate that the calls to library functions in `angr`'s `SimProcedures` are only approximations, which can reduce the completeness of the analysis. It is essential to note that system calls are implemented as `SimProcedures`. Unfortunately, some system calls have not yet been implemented in `angr`, further limiting the accuracy and completeness of the analysis. It is crucial to address these inaccuracies and implement missing system calls in the `SimProcedures` to improve the overall effectiveness of `angr`.

4.3.2 Symbolic Memory Model

The default memory model used by `angr` is inspired by Mayhem [10]. This memory model supports limited symbolic reads and writes. If the memory index of a read is symbolic and the range of possible values of this index is too wide, the index is concretized to a single value. Similarly, if the memory index of a write is symbolic at all, the index is concretized to a single value. This means that if a UAF vulnerability depends on a specific symbolic read or write, but the memory access is concretized to a different value, it cannot be detected.

4.3.3 Symbolic Lengths in the Heap Model

Another factor that impacts the completeness of the UAF analysis is the heap model. The heap implementation of `angr` uses a model that concretizes all symbolic allocation sizes by silently maximizing their potential size up to the maximum variable size, as specified in the libc plugin. This trade-off attempts to solve the decision of how much space to allocate upon request for a symbolic size. The silent maximization approach has benefits, as the explored state space will not be constrained. This could sometimes work when an allocation is returned smaller than requested, but the program only uses it partially. This lack of fidelity may however lead to the program overwriting other allocations, as it should assume the allocation is as large as requested. As a result, although not evident in the real-world evaluation, false positives remain possible. Furthermore, paths that expect the allocation size to be larger cause the analysis to miss the opportunity to explore those paths. Consequently, all UAF vulnerabilities that depend on paths unexplorable due to this limitation or expect the allocation to be larger than the maximum value cannot be detected.

5 Related Work

This chapter provides an overview of the latest research in detecting UAF vulnerabilities. The discussion primarily focuses on two key subtopics: UAF detection techniques and the integration of static analysis and symbolic execution.

5.1 Use-After-Free Detection

The paper [24] provides a thorough analysis, comparison, and evaluation of current UAF detection techniques. Experimental results show that static UAF detectors effectively identify intra-procedural UAFs but struggle with detecting inter-procedural UAFs in real-world programs. This suggests that static analysis tools may not wholly uncover all potential vulnerabilities, especially those spanning multiple procedures. As a result, dynamic UAF detectors remain the preferred choice for detecting inter-procedural UAFs. The paper concludes that the performance of state-of-the-art static detection approaches still needs improvement. Although the fundamental limitations of static analysis cannot be eliminated, the paper proposes that some disadvantages of existing static UAF detectors could be mitigated by integrating static analysis with symbolic execution and machine learning techniques.

5.1.1 Static Analysis

The paper [41] presents a scalable static analysis technique for detecting UAF and double-free vulnerabilities in binary code. This method combines CFG construction, alias analysis, and pointer tracking. Instead of using in-lining for inter-procedural analysis, the approach employs function summaries, which help prevent redundant analysis and reduce the performance overhead. The authors have developed a prototype called UAFDetector and tested its effectiveness using the Juliet Test Suite and real-world programs. In the Juliet Test Suite, UAFDetector achieved a false negative rate of 2.39% and a zero false positive rate. When evaluated on real-world programs, the tool had more than twice as many false positives as true positives. However, it successfully detected 6 out of 7 CVEs.

Tac [39] is a machine learning-guided static UAF vulnerability detection framework. It learns the correlations between program features and UAF-related aliases using a support vector machine and exploits this knowledge to improve the precision of the alias analysis. Nevertheless, the approach is not yet sound and requires many marked training samples.

CRed, a static analysis technique introduced in [40], is implemented in LLVM to identify UAF vulnerabilities in C source code through pointer analysis. Using a spatiotemporal context reduction approach, CRed minimizes the exponential number of contexts. It focuses on those relevant for analyzing pairs of free and use sites, enabling it to handle larger codebases efficiently. Notably, CRed successfully detects all UAF vulnerabilities in the old Juliet Test Suite 1.2, consisting of only 138 test cases, and uncovers new vulnerabilities in real-world C applications.

5.1.2 Dynamic Analysis

[8] presents Undangle, a dynamic runtime approach that aims to detect and diagnose UAF and double-free vulnerabilities early on. This approach employs a combination of taint analysis and pointer tracking to identify the creation of dangling pointers rather than focusing solely on their dereferencing. This methodology effectively minimizes the likelihood of incomplete fixes, as it automatically gathers information on all dangling pointers involved in the vulnerability. However, since Undangle relies on analyzing execution traces, it has a significant performance overhead and is unsuitable for analyzing large-scale programs.

5.2 Combining Static Analysis and Symbolic Execution

The approach of using Symbolic Execution to check the results of static analyses to reduce false positives is not new.

[28] developed SATracer in LLVM, a tool designed to enhance the results of static analysis tools by integrating symbolic execution techniques. SATracer automatically searches for vulnerable paths in a given defect report, including a starting point (source), a bug-exposing statement (sink), and crucial steps to activate the bug's root cause. The tool then extracts a step-by-step path with a high likelihood of triggering the reported defect, considering it an actual bug if a corresponding vulnerable path is identified. SATracer was evaluated using the Juliet Test Suite and applied to 21 real-world applications. The results are promising, with SATracer successfully removing 71.4% of false alarms reported by ClangSA in 10 hours while

confirming 29 actual UAF bugs and 895 real null-pointer-dereference bugs. However, the results should be viewed critically, as SATracer is not open-source, had five false negatives on the Juliet Test Suite, and the versions and CVEs of the real-world applications were not provided, making the results non-reproducible. Moreover, SATracer cannot work with binary files, since it is written in LLVM and needs the source code.

The paper [19] introduces a fully automated technique to detect and trigger UAF vulnerabilities in binary code using a novel method called Weighted Slice-Guided Dynamic Symbolic Execution (WS-Guided DSE). This approach combines static analysis, which relies on the GUEB tool [20], and dynamic symbolic execution (DSE) performed on the BINSEC/SE platform [14]. The static analysis identifies a UAF and extracts a slice containing it. This slice is weighted and utilized to guide the DSE exploration. Once a UAF is validated, a proof-of-concept (PoC) is generated. As an illustration, the authors provide a detailed PoC exploit that triggers a previously unknown vulnerability in JasPer, leading to the CVE-2015-5221. However, this CVE is a double-free, which is not addressed by the approach in this thesis, so the results cannot be compared directly. Moreover, Feist et al. recognize the need to apply their methodology to a larger dataset to evaluate its capabilities better.

In the paper [37], the authors introduce Arbiter, a tool for discovering vulnerabilities in binary programs. The tool aims to bridge the gap between static and dynamic analysis techniques by identifying specific vulnerability properties that enhance both static and dynamic vulnerability detection. This enables the effective combination of static analysis and UCSE for precise vulnerability detection while maintaining scalability. Arbiter also utilizes the `angr` framework for symbolic execution on binaries and is designed to make the creation of new vulnerability classes inexpensive. The tool was evaluated by analyzing 76,516 x86-64 binaries from the Ubuntu repositories for four Common Weakness Enumerations (CWEs): CWE-131, CWE-252, CWE-134, and CWE-337. The evaluation discovered new vulnerabilities, including a flaw introduced into programs during compilation. While the results are noteworthy, it should be noted that the vulnerabilities used in the evaluation are simpler than UAF vulnerabilities and do not require interprocedural analysis. Nonetheless, Arbiter achieved impressive results even with the use of UCSE.

6 Conclusion

In this thesis, a symbolic execution analysis was developed using the `angr` framework to identify Use-After-Free (UAF) vulnerabilities in binary files. The analysis incorporates Directed Symbolic Execution, Under-Constrained Symbolic Execution (UCSE), and Veritesting techniques to address the path explosion issue. The effectiveness of this approach in detecting UAF vulnerabilities was evaluated using the Juliet Test Suite and four real-world binaries with documented UAF CVEs. The evaluation results for the Juliet Test Suite demonstrated that the proposed approach could successfully identify UAF vulnerabilities in small, synthetic binaries. However, the analysis did not report any UAFs in the real-world binaries, primarily due to scalability issues caused by path explosion and reduced completeness resulting from inaccuracies in `angr`'s library function simulations and heap modeling. The assessment of real-world binaries revealed that Directed Symbolic Execution, combined with the results of a static analysis tool, holds significant potential for mitigating the path explosion issue. Nevertheless, applying UCSE on every function is not scalable, and `angr`'s Veritesting implementation suffers from memory exhaustion.

Future research should focus on closely integrating static analysis results with symbolic execution. This integration would involve using the same CFG for both the static tool and Directed Symbolic Execution and incorporating information on dangling pointer creation, not just the dereferencing location. Furthermore, future work should address infinite concrete loops and enhance the utilization of UCSE by initiating the process at the function responsible for creating dangling pointers.

References

[1] Afek, Jonathan & Sharabani, Adi: *Dangling pointer: Smashing the pointer for fun and profit*. 2007.

[2] Andriesse, Dennis: *Practical binary analysis: build your own Linux tools for binary instrumentation, analysis, and disassembly*. no starch press, 2018.

[3] *angr*. url: https://angr.io/ (visited on 04/05/2023).

[4] Assured Software, Center for: *Juliet Test Suite v1.2 for C/C++ User Guide*. https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf. [Online; accessed 4-March-2023]. 2012.

[5] Avgerinos, Thanassis; Rebert, Alexandre; Cha, Sang Kil & Brumley, David: "Enhancing symbolic execution with veritesting". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 1083–1094.

[6] Baldoni, Roberto; Coppa, Emilio; D'elia, Daniele Cono; Demetrescu, Camil & Finocchi, Irene: "A survey of symbolic execution techniques". In: *ACM Computing Surveys (CSUR)* 51.3, 2018, pp. 1–39.

[7] Black, Paul E: *Juliet 1.3 test suite: changes from 1.2*. en. Tech. rep. NIST TN 1995. Gaithersburg, MD: National Institute of Standards and Technology, 2018, NIST TN 1995. url: http://nvlpubs.nist.gov/nistpubs/TechnicalNotes/NIST.TN.1995.pdf (visited on 04/27/2023).

[8] Caballero, Juan; Grieco, Gustavo; Marron, Mark & Nappa, Antonio: "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 2012, pp. 133–143.

[9] Cadar, Cristian & Sen, Koushik: "Symbolic execution for software testing: three decades later". In: *Communications of the ACM* 56.2, 2013, pp. 82–90.

[10] Cha, Sang Kil; Avgerinos, Thanassis; Rebert, Alexandre & Brumley, David: "Unleashing mayhem on binary code". In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394.

[11] Cowan, Crispin; Wagle, F; Pu, Calton; Beattie, Steve & Walpole, Jonathan: "Buffer overflows: Attacks and defenses for the vulnerability of the decade". In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. Vol. 2. IEEE. 2000, pp. 119–129.

[12] *CWE - CWE-416: Use After Free (4.10)*. url: https://cwe.mitre.org/data/definitions/416.html (visited on 04/04/2023).

[13] *cwe_checker*. url: https://github.com/fkie-cad/cwe_checker (visited on 05/03/2023).

[14] David, Robin; Bardin, Sébastien; Ta, Thanh Dinh; Mounier, Laurent; Feist, Josselin; Potet, Marie-Laure & Marion, Jean-Yves: "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 653–656.

[15] De Moura, Leonardo & Bjørner, Nikolaj: "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer. 2008, pp. 337–340.

[16] Emanuelsson, Pär & Nilsson, Ulf: "A comparative study of industrial static analysis tools". In: *Electronic notes in theoretical computer science* 217, 2008, pp. 5–21.

[17] Engler, Dawson & Dunbar, Daniel: "Under-constrained execution: making automatic code destruction easy and scalable". In: *Proceedings of the 2007 international symposium on Software testing and analysis*. 2007, pp. 1–4.

[18] Ernst, Michael D: "Static and dynamic analysis: Synergy and duality". In: *WODA 2003: ICSE Workshop on Dynamic Analysis*. 2003, pp. 24–27.

[19] Feist, Josselin; Mounier, Laurent; Bardin, Sébastien; David, Robin & Potet, Marie-Laure: "Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free". In: *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. 2016, pp. 1–12.

[20] Feist, Josselin; Mounier, Laurent & Potet, Marie-Laure: "Statically detecting use after free on binary code". In: *Journal of Computer Virology and Hacking Techniques* 10.3, 2014, pp. 211–217.

[21] *GitHub - angr/angr: A powerful and user-friendly binary analysis platform!* URL: https://github.com/angr/angr (visited on 05/03/2023).

[22] Goseva-Popstojanova, Katerina & Perhinschi, Andrei: "On the capability of static code analysis to detect security vulnerabilities". In: *Information and Software Technology* 68, 2015, pp. 18–33.

[23] *Gotchas when using angr - angr documentation*. URL: https://docs.angr.io/en/latest/advanced-topics/gotchas.html (visited on 05/12/2023).

[24] Gui, Binfa; Song, Wei; Xiong, Hailong & Huang, Jeff: "Automated Use-After-Free Detection and Exploit Mitigation: How Far Have We Gone?" In: *IEEE Transactions on Software Engineering* 48.11, 2021, pp. 4569–4589.

[25] *Juliet C/C++ 1.3*. en. URL: https://samate.nist.gov/SARD/test-suites/112 (visited on 04/22/2023).

[26] Landi, William: "Undecidability of static analysis". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4, 1992, pp. 323–337.

[27] Lee, Byoungyoung; Song, Chengyu; Jang, Yeongjin; Wang, Tielei; Kim, Taesoo; Lu, Long & Lee, Wenke: "Preventing Use-after-free with Dangling Pointers Nullification." In: *NDSS*. Citeseer. 2015.

[28] Li, Guangwei; Yuan, Ting; Lu, Jie; Li, Lian; Zhang, Xiaobin; Song, Xu & Zhang, Kejun: "Exposing Vulnerable Paths: Enhance Static Analysis with Lightweight Symbolic Execution". In: *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2021, pp. 441–451.

[29] Ma, Kin-Keung; Yit Phang, Khoo; Foster, Jeffrey S & Hicks, Michael: "Directed symbolic execution". In: *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18*. Springer. 2011, pp. 95–111.

[30] Meng, Xiaozhu & Miller, Barton P: "Binary code is not easy". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016, pp. 24–35.

[31] *NVD - Statistics*. URL: https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&search_type=all&cwe_id=CWE-416&isCpeNameSearch=false (visited on 05/08/2023).

[32] Ramos, David A & Engler, Dawson: "Under-constrained symbolic execution: Correctness checking for real code". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 49–64.

[33] Schwartz, Edward J; Avgerinos, Thanassis & Brumley, David: “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)”. In: *2010 IEEE symposium on Security and privacy*. IEEE. 2010, pp. 317–331.

[34] Shoshitaishvili, Yan et al.: “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *IEEE Symposium on Security and Privacy*. 2016.

[35] Thompson, Ken: “Reflections on trusting trust”. In: *Communications of the ACM* 27.8, 1984, pp. 761–763.

[36] *throwing a tantrum, part 1: angr internals*. url: https://angr.io/blog/throwing_a_tantrum_part_1/ (visited on 04/26/2023).

[37] Vadayath, Jayakrishna et al.: “Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 413–430.

[38] Xu, Wen; Li, Juanru; Shu, Junliang; Yang, Wenbo; Xie, Tianyi; Zhang, Yuanyuan & Gu, Dawu: “From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 414–425.

[39] Yan, Hua; Sui, Yulei; Chen, Shiping & Xue, Jingling: “Machine-learning-guided typestate analysis for static use-after-free detection”. In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. 2017, pp. 42–54.

[40] Yan, Hua; Sui, Yulei; Chen, Shiping & Xue, Jingling: “Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities”. In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 327–337.

[41] Zhu, Kailong; Lu, Yuliang & Huang, Hui: “Scalable static detection of use-after-free vulnerabilities in binary code”. In: *IEEE Access* 8, 2020, pp. 78713–78725.

List of Figures

List of Tables

List of Listings