POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

ByteMatcher: a tool for semantic equivalence of bytecode through symbolic execution

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Lorenzo Fratus**

Student ID: 962710
Advisor: Prof. Mario Polino
Co-advisors: Armando Bellante, Lorenzo Binosi
Academic Year: 2021-22

Abstract

Automated analysis of computer programs is a powerful tool in the arsenal of software developers. In the last decades, researchers presented numerous techniques to extract useful information from software. Some examples are tools to generate high-coverage tests, analyze execution flows to find unreachable code or detect vulnerabilities from known patterns. Nevertheless, being able to check whether two pieces of code are semantically equivalent is still an open challenge. This thesis presents ByteMatcher, a tool that employs the dynamic analysis technique known as symbolic execution to decide on the semantic equivalence of two bytecodes (fragments of compiled programs). This problem is generally undecidable, however it becomes solvable when faced under specific assumptions. ByteMatcher implements state-of-the-art approaches to face the four main challenges of symbolic execution: symbolic memory accesses, environment interactions, state space explosion, and symbolic constraints resolution. The most innovative contribution of this work is a technique that allows developers to compare the results of the execution of two bytecodes to know whether they are semantically equivalent. Developers have complete control over the sensitivity of ByteMatcher and can choose among four levels of equivalence to affect how the algorithm performs the comparison. On top of that, ByteMatcher can detect (possible) buffer overflow vulnerabilities by analyzing patterns in memory accesses during the symbolic execution of a bytecode. ByteMatcher was tested on bytecodes sampled from real-world programs. In particular, to validate the ability to detect semantic equivalence, ByteMatcher performed more than 160 000 comparisons extracted from 431 GitHub repositories. ByteMatcher can distinguish between equivalent and non-equivalent bytecodes with 81.16% of accuracy and 78.71% of F1-score. To test the functionality of buffer overflow detection, ByteMatcher executed more than 20 000 bytecodes sampled from the 116 challenge binaries of the DARPA's CGC qualifying event. This experiment showed that ByteMatcher can detect 5 different types of vulnerabilities (according to the CWE standard), providing predictions with 66.38% of accuracy and 65.49% of F1-score.

Keywords: semantic equivalence, buffer overflow, symbolic execution

Abstract in lingua italiana

L'analisi automatizzata di programmi informatici è un potente strumento nell'arsenale degli sviluppatori software. Negli ultimi decenni, i ricercatori hanno presentato numerose tecniche per estrarre informazioni utili dai software. Alcuni esempi sono strumenti per generare test ad alta copertura, analizzare flussi di esecuzione per trovare codice irraggiungibile o rilevare vulnerabilità a partire da pattern noti. Tuttavia, essere in grado di verificare se due codici sono semanticamente equivalenti è ancora una sfida aperta. Questa tesi presenta ByteMatcher, uno strumento che impiega la tecnica di analisi dinamica nota come esecuzione simbolica per decidere l'equivalenza di due bytecode (frammenti di programmi compilati). Questo problema è generalmente indecidibile, tuttavia diventa risolvibile se affrontato sotto ipotesi specifiche. Questo strumento implementa approcci all'avanguardia per affrontare le quattro sfide principali dell'esecuzione simbolica: accessi simbolici alla memoria, interazioni con l'ambiente, esplosione dello spazio degli stati e risoluzione dei vincoli simbolici. Il contributo più innovativo di questa tesi è una tecnica che consente agli sviluppatori di confrontare i risultati dell'esecuzione di due bytecode per sapere se sono equivalenti. Gli sviluppatori hanno il completo controllo sulla sensibilità di ByteMatcher e possono scegliere tra quattro livelli di equivalenza per influenzare il modo in cui l'algoritmo esegue il confronto. Inoltre, ByteMatcher può rilevare (possibili) vulnerabilità di buffer overflow analizzando i pattern negli accessi in memoria durante l'esecuzione simbolica di un bytecode. ByteMatcher è stato testato su bytecode campionati da programmi reali. In particolare, per convalidare la capacità di rilevare l'equivalenza semantica, ByteMatcher ha eseguito più di 160 000 confronti generati da 431 repository GitHub. ByteMatcher può distinguere tra bytecode equivalenti e non equivalenti con 81.16% di accuratezza e 78.71% di F1-score. Per testare la funzionalità di rilevamento dei buffer overflow, ByteMatcher ha eseguito più di 20 000 bytecode presi dai binari della fase di qualifica del CGC organizzato dalla DARPA. Questo esperimento ha dimostrato che ByteMatcher può rilevare 5 diversi tipi di vulnerabilità (secondo lo standard CWE), fornendo previsioni con 66.38% di accuratezza e 65.49% di F1-score.

Parole chiave: equivalenza semantica, buffer overflow, esecuzione simbolica

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 Motivation **3**
1.1 Problem Statement . . . 3
1.2 State of the Art . . . 4
1.2.1 Symbolic Execution . . . 4
1.2.2 Memory Models . . . 6
1.2.3 Loop Summarization . . . 7
1.2.4 Binary Similarity . . . 8
1.3 Goals and Challenges . . . 9

2 Background **11**
2.1 Code Lifting . . . 11
2.2 Symbolic Variables . . . 12
2.3 Symbolic Loop Summaries . . . 12
2.3.1 Glossary and Definitions . . . 13
2.3.2 Loop Summary Generation . . . 14

3 Approach **17**
3.1 Algorithm Overview . . . 17
3.1.1 Symbolic Execution . . . 17
3.1.2 Semantic Equivalence . . . 19
3.1.3 Buffer Overflow Detection . . . 19
3.2 Approach Details . . . 19

3.2.1 Fully-Symbolic Memory Model 19
3.2.2 Uninterpreted Function Calls 25
3.2.3 Comparison and Equivalence Levels 26
3.2.4 Buffer Overflow Detection 27

4 Implementation Details 29
4.1 System Architecture 29
4.1.1 Tools and Technologies 29
4.1.2 Project Structure 30
4.2 System Details 32
4.2.1 Executor 32
4.2.2 ExecutorState, RegisterState, and MemoryState 34
4.2.3 LoopSummary 37
4.2.4 Additional Features 40

5 Experimental Validation 43
5.1 Goals of the Validation 43
5.2 Experimental Setup 43
5.2.1 Environment 44
5.2.2 Testing Approach 44
5.3 Semantic Equivalence 45
5.3.1 Goal and Strategy 45
5.3.2 Dataset 45
5.3.3 Results 47
5.4 Buffer Overflow Detection 52
5.4.1 Goal and Strategy 52
5.4.2 Dataset 52
5.4.3 Results 54

6 Limitations 57

7 Future Works 59

8 Conclusions 61

Bibliography 63

List of Algorithms 67

List of Figures 69

List of Tables **71**

Acknowledgements **73**

Introduction

Computer science is at the foundation of modern society, in which technology plays a crucial role in every aspect of people's life. This results in the release of an incredible number of software programs every year. Despite all the best practices and principles developers follow when producing software, writing bug-free code on the first try remains an open challenge. On the other hand, designing tests that correctly identify every error and vulnerability is a notoriously complex task.

For this reason, researchers developed tools like KLEE [6] for automated test generation or Angr [21] for binary code analysis, arousing widespread interest in the scientific community. Regardless of their excellent performance in the respective tasks and their popularity among researchers, these tools have more than one limitation, mainly related to well-known challenges in symbolic execution.

This thesis presents ByteMatcher, a tool based on the dynamic analysis of bytecodes (also known as shellcodes). A bytecode is any sequence of bytes (not necessarily a complete function or program) that a processor can interpret as instructions. These strings of bytes derive from a compilation process and represent software in memory.

This work takes inspiration from the implementation of Angr and improves it by introducing, among other things, a fully-symbolic memory model. This model represents the machine's state inside of ByteMatcher and can handle memory operations on symbolic values without concretization or path splitting.

On top of that, ByteMatcher implements an algorithm to handle input-dependent loops by producing a reusable summary of their bodies from just a handful of iterations. The introduction of a summary limits the path explosion problem that is still today a concern for symbolic execution.

During the process of loop summarization, ByteMatcher can also detect specific indicators in the memory model of the program under execution that may hint at a possible buffer overflow vulnerability. This characteristic is a stepping stone to introduce ByteMatcher to vulnerability detection through dynamic analysis.

Lastly, this thesis gives its main contribution with a mechanism to compare two bytecodes' execution and predict whether they are semantically equivalent. Semantic equivalence is the ability of two pieces of code (differing from each other) to produce the same effects on the state of a machine upon execution.

The results of the experimental evaluation of the first implementation of ByteMatcher are positive. The data supports ByteMatcher's ability to recognize semantically equivalent bytecodes correctly and shows that it can detect specific types of vulnerabilities correlated to buffer overflows. Overall, the experiments demonstrate the validity of the techniques presented in this research and the approach used to condense them into a single tool for symbolic execution. The collected data can lead the way for future research to improve the performance of the current implementation and enhance it with new features.

To summarize, the main contributions of this thesis are:

- ByteMatcher, a tool for symbolic execution with a fully-symbolic memory model.
- A technique for the summarization of input-dependent loops.
- An approach to decide on the semantic equivalence of two bytecodes.
- A mechanism to detect (possible) buffer overflows based on dynamic analysis.

1 | Motivation

1.1. Problem Statement

Software development and cybersecurity are two fields that play a vital role in today's world in which technology permeates essential aspects of people's lives, like work, education, finance, and healthcare. For this reason, it is crucial to focus on devising new tools and techniques to help developers increase computer programs' security.

Given these circumstances, developers and security experts would benefit from the ability to automatically test the semantic equivalence of two bytecodes. A bytecode is a fragment of a computer program, stripped of all symbols, composed of a series of instructions in machine code. A tool that can reliably perform such a task could significantly impact fields like vulnerability assessment, regression verification, and shellcode generation.

Whenever a previously unknown vulnerability emerges in popular libraries and frameworks, security experts must check existing programs to find any pattern that matches that discovery. In this case, a tool that can automatically analyze functions to detect equivalence with the vulnerable sample would make the process efficient and exhaustive by checking for any possible variation of the offending code.

After updating a piece of code, for example, with optimization or refactoring, software developers must perform regression tests to ensure the overall functionality is untouched. These tests usually check that the function produces the same output for a strategically-chosen set of inputs. A tool for semantic equivalence can perform regression verification without any additional input from the developers by checking that the code before the update matches the one after the update.

Finally, penetration testing is one of the most effective ways to test the resilience of a computer program to external attacks. Penetration tests consist of professional hackers trying to break into the systems under test in a controlled and systematic way. Sometimes, hackers use shellcodes to exploit vulnerabilities (i.e., bytecodes that perform malicious actions like opening a shell). Starting from existing exploits, hackers can use a tool to

produce new, semantically equivalent shellcodes that meet specific constraints, like limited size or absence of special bytes.

This research presents the steps that led to the development of ByteMatcher, a tool that employs dynamic analysis techniques, symbolic execution in particular, to decide whether two bytecodes are semantically equivalent.

1.2. State of the Art

This Section presents and briefly analyzes the state of the art concerning each topic this work touches. This breakdown helps to illustrate and categorize the fields to which this research contributes.

1.2.1. Symbolic Execution

The authors of [2] say that symbolic execution is "a popular analysis technique introduced in the mid-'70s to test whether certain properties can be violated by a piece of software". This technique involves analyzing a program using a set of logical constraints that describe the characteristics of the input rather than concrete, well-formed values. In this way, it is possible to explore a variety of execution paths simultaneously, delegating to a constraint solver the construction of the set of inputs that cause a property violation when detected. In addition to introducing the concept of symbolic execution, the survey provides "an overview of the main ideas, challenges, and solutions developed in the area, distilling them for a broad audience". The article outlines four challenges that every symbolic executor has to face explaining how they generally require some assumptions depending on the context.

- Memory: how to handle pointers, arrays, and complex objects? How to work with symbolic addresses?
- Environment: how to handle the interaction with library or system code and the resulting side effects?
- State space explosion: how to prevent path explosion, for example, in loops?
- Constraint solving: what does a constraint solver do?

Despite appearing for the first time in an article of 2008, KLEE [6] is still a prevalent tool in symbolic execution that focuses on automatic high-coverage test generation and bug finding in general. The tool uses a heuristic approach to choose which path to explore to maximize the coverage and can report a set of bug-triggering inputs and signal dangerous

operations. KLEE was tested on the programs of the GNU Coreutils and Busybox suites to obtain an average coverage above 90% per tool. The paper tackles two of the challenges presented before. In particular, when cloning a state, the authors mitigate the path explosion problem using a compact and efficient state representation and various optimizations like the copy-on-write mechanism for the content of the memory. They also implement custom handlers to mock the functionality of various system calls to manage the environment issue. KLEE also implements a broad set of optimizations to simplify logical expressions and speed up the solver's work.

Presented during the same year by researchers from Microsoft, SAGE [18] is a whitebox fuzz testing tool that uses symbolic execution to generate tests. The tool executes a program starting from a well-formed input, gathering constraints from conditional statements along the way. To generate new inputs for subsequent runs, SAGE negates each constraint individually. SAGE implements a heuristic that maximizes the code coverage to select the following input to test. However, the authors explain that full coverage, even if theoretically possible, is generally improbable due to the cardinality of the input space. On the other hand, executing a program based on a concrete input solves most of the challenges presented before.

Another research introduces Mayhem [7], a tool that finds exploitable bugs in a binary program employing symbolic execution. An attractive characteristic of this tool is that it provides a working shell-spawning exploit with every bug it reports. As for the challenges, Mayhem tries to solve the path explosion problem by implementing a hybrid (concolic) execution technique. The execution starts with an online approach where the path branches at every conditional statement (like KLEE [6]). As soon as the memory usage reaches a fixed threshold, Mayhem moves to an offline approach that executes only one path at a time (like SAGE [18]). Mayhem also implements an index-based memory model which balances efficiency and completeness by concretizing write operations and allowing reads to be symbolic.

Finally, [21] presents a unifying framework for binary analysis named Angr. Angr implements several well-known analysis techniques in a systemized way that allows comparing different approaches and evaluating their advantages and disadvantages. All the analyses implemented by Angr are cross-architecture and cross-platform. The main idea of the authors of this paper is to create a framework that other developers can use as a starting point to create next-generation binary analysis techniques. For this reason, their code is open source and available to the community. Angr also implements approaches presented in other articles to improve symbolic execution. Two notable examples are the index-based memory model presented in Mayhem [7] and the Veritesting technique devel-

oped for another framework called MergePoint [1]. Veritesting introduces a mechanism to automatically switch between dynamic (path-based) and static (statement-based) symbolic execution, mitigating the problems of path explosion and solver blowup.

1.2.2. Memory Models

Defining the correct memory model for each task is one of the main challenges of symbolic execution. The ideal memory model should be able to perfectly handle any memory operation, even with symbolic addresses, in a quick way and keep the size of the representation small. Even if this model is only theoretical, the last decade brought a series of exciting options, each with advantages and disadvantages.

According to the authors, the paper at [23] presents the first memory model to support fully-symbolic addresses. The article presents a segment-offset-plane memory model that stores addresses as *segment:offset* pairs and divides the memory content into *planes* according to their semantic information. This model is part of an open-source project named Bugst. On write operations, it splits data between the *boostMap* that keeps track of concrete writes and the *iteList* used to record all the symbolic writes. When reading the data, it directly returns the value if the address matches an entry in the *boostMap*. Otherwise, the model loops on the *iteList* to construct a recursive If-Then-Else (ITE) formula listing all the entries whose address collides with the one being read (from the youngest to the oldest). The authors outline some of the limitations of this approach, particularly the fact that this is a fragmented model (due to the structure of the memory) and that it never removes records from the *iteList* even when newer entries overwrite them. While the former is intrinsic in the approach, the latter would be resolvable by checking collisions when appending a new record to the list.

The memory model presented in [15] handles symbolic pointers by decomposing the memory in non-overlapping segments so that every pointer only refers to objects in a single segment, bypassing the need for forking the execution. The model identifies abstract memory objects by their static allocation point in the program. Each symbolic pointer is mapped to a set of objects using a (conservative) *point-to-point* analysis. To improve the efficiency, the authors set a maximum size for segments, meaning that large objects may span multiple segments and cause forking when accessed. This model, implemented for KLEE [6], is available as an open-source project.

The paper at [22] extends the previous solution with a relocatable addressing model to improve memory objects. This version uses symbolic expressions to represent the address of each object instead of concrete addresses. In this way, it is possible to relocate them

by changing the symbolical expression. In detail, resolving a symbolic pointer creates a new segment containing all the referred objects. Moreover, big memory objects are divided into smaller adjacent objects to limit the size of the SMT array. Unfortunately, as emphasized by the authors in the papers [15, 22], the performance of this algorithm heavily relies on the precision of the point-to-point analysis that can make the approach degenerate in either a flat or a forking memory model.

The article at [4] contributes by proposing a systemized overview of the memory model employed by four mainstream symbolic execution frameworks and presents a new approach named MemSight (already introduced in [8]). MemSight aims to reduce the need for concretization by directly using symbolic expressions to address data, maintaining the possibility to access all the possible alternative states resulting from symbolic memory operations. The model handles symbolic write operations by inserting a tuple in a memory-wide paged interval tree and produces optimized ITE expressions on read operations. It also handles concrete writes in a paged hashmap with copy-on-write to improve the model's performance. MemSight accounts for state merging (a technique that symbolic executors wildly use to reduce the number of states) by proposing algorithms to join the content of two states that share a common ancestor. The paper also presents a set of refinements and optimizations, including employing symbolic values to represent read operations on uninitialized memory. The authors provide an open-source implementation of MemSight for both Angr [21] and KLEE [6].

The last model in this analysis is MInt [5], an extension of MemSight [4, 8], which introduces algorithms to handle memory-intensive operations such `memset` and `memcpy`. MInt extends the original algorithms to work with two new sets of tuples. The tuples track the symbolic range that each operation is targeting and allow the model to reason on the effects of the operation only when a read on an address in that area occurs. Unlike its predecessor, MInt is available only for Angr.

1.2.3. Loop Summarization

Managing symbolic loops is crucial in symbolic execution to avoid the problem of path explosion. Typical frameworks handle loops using concretization or limiting the executions of the same code section by discarding additional iterations. This approach might lose precious information and miss possible bugs or vulnerabilities that appear after many loop iterations (e.g., buffer overflows). In recent years, researchers developed some approaches to try to mitigate this loss. This research analyzes the one that looks more promising, given its primary goal: loop summarization. Summarizing a loop means producing a

logical (symbolic) formula that describes the effects of the loop execution and is parametric in the number of iterations. The executor can then apply this formula to a symbolic state to obtain the same result without the effort of symbolically executing the loop.

Proteus [24, 25], is a loop analysis framework that aims to produce a disjunctive loop summary for multi-path, possibly nested, loops. Depending on the loop behavior, the authors classify loops into four types and define which type Proteus can summarize precisely and which with an approximation. The summarization uses a Path Dependency Automaton (PDA) extracted from the loop's Control Flow Graph (CFG). The algorithm systematically traverses the PDA to account for every execution path possible, summarizing variables that follow complex sequences (constant, geometric, or combined). If the approach looks very promising, the fact that its implementation is private and that the papers need to be more specific to be used to replicate their results makes it impossible to consider it for ByteMatcher.

The article at [12] presents an approach to (partially) summarize loop-invariant generation that, under defined circumstances, can produce a summary of the loop itself. The loop structure of the program is discovered dynamically during a single symbolic execution. The algorithm uses loop-guard pattern-matching rules to estimate the number of iterations of input-dependent loops. A summary in the form of the logic formula $pre_{loop} \wedge post_{loop}$ is obtained whenever the execution confirms an estimation. pre_{loop} is a set of conditions describing the executions covered by the summary, and $post_{loop}$ is a set of conditions that (partially) captures the side effects of the loop. This approach produces a partial summary because it can detect and summarize only induction variables (i.e., variables modified by a constant value during each execution), possibly losing the effects of the loop on other variables. The authors implement the technique for SAGE [18], but the code is not publicly available. Nevertheless, the article extensively explains the algorithm and provides a pseudo-code implementation.

1.2.4. Binary Similarity

Lastly, this Subsection illustrates two existing static analysis techniques to evaluate the similarity of binaries. This work is relevant as the main contributions of these papers are approaches that decide if two functions are similar by only considering their compiled form. This idea aligns with the primary goal of this thesis, but the approaches and fields of application differ vastly.

The authors of [16] firstly present pre-existing techniques that compute binary similarity based on function embeddings. The idea behind these approaches is to transform

the binaries into multi-dimensional vector representations (embeddings) that the tool can compare using efficient geometric operations. The main shortcomings of these solutions are that they require manually-selected features (that can introduce bias in the resulting embedding), assume the presence of symbols in the binary, and work only on specific CPU architectures. The article then presents SAFE, a new technique to generate embeddings using a self-attentive neural network that overcomes the previous limitations. This technique focuses only on a specific case of similarity: two functions are similar if compiled from the same source.

A different approach presented in [10] aims to recognize the similarity of stripped binaries compiled using different compilers or optimization levels and for multiple architectures. The main idea is to decompose the code into small comparable fragments that preserve the dependence chain, called strands. Each strand is re-optimized using the compiler optimizer to transform it into a canonical, normalized form that the algorithm efficiently compares using a hash function. The algorithm employs a statistical framework built using samples *from the wild* that focuses on comparing statistically significant strands. A tool called GitZ implements this technique. Even in this circumstance, the approach recognizes two functions as similar only if they originate from the same code.

1.3. Goals and Challenges

This thesis makes an effort to condense the best solutions and techniques from multiple research fields in a comprehensive tool that can test the semantic equivalence of two bytecodes using symbolic execution. Achieving this goal is more complex than one might expect. In fact, from the analysis of the current research landscape, no article describes a tool or technique that can perform a similar task.

Moreover, comparing two computer programs is closely related to the *halting problem*, a fundamental problem in computer science that wonders if it is possible to establish whether an algorithm ends its computation or loops indefinitely. Researchers have proven that no general solution can solve the halting problem for every input. Therefore, it is no surprise that some bytecodes might not be analyzable by ByteMatcher.

In order to accomplish the primary goal of this work, the implementation of ByteMatcher takes inspiration from state-of-the-art approaches by picking the most promising and suitable solution from each area of research. In particular, ByteMatcher contributes by implementing a symbolic executor that can dynamically analyze the behavior of fragments of computer programs, stripped of all symbols, composed of a series of instructions in machine code, known as bytecodes. This executor employs a fully-symbolic

memory model to abstract the machine's state that simulates the code.

Additionally, ByteMatcher implements an algorithm to compare the results of two symbolic executions to establish whether the source bytecodes are semantically equivalent, meaning whether they produce identical modifications to the state of the machine when fed with the same input.

Finally, ByteMatcher exploits existing research results to implement a technique to deal with symbolic loops that are known to be hard to execute symbolically without running into the phenomenon of path explosion.

2 | Background

This Chapter introduces the theoretical concepts behind the techniques employed by ByteMatcher. It presents a few terms and notions that assist the reader in understanding the following Chapters. Refer to Chapter 3 for the description of the approach used or Chapter 4 for the technical details of the implementation of ByteMatcher.

2.1. Code Lifting

The term code lifting refers to the translation of a computer program from its binary representation into an Intermediate Language (IL). An IL resembles the assembly language and abstracts the machine code as a series of (complex) human-readable instructions that capture all the information included in the original program. The abstraction provided by the IL is essential to analyze and simulate the code at the logical level using another computer program.

ByteMatcher employs VEX, the IL used in Angr [21] (that in turn borrowed it from Valgrind [19]). VEX is architecture-agnostic, meaning it can represent code compiled from various architectures, ignoring specific details like the name and number of registers or the endianness and segmentation of the memory. Moreover, every VEX instruction is side-effect-free. The lifter (the program that performs the code lifting) encodes every operation as a series of independent VEX instructions. For example, `push` and `pop` produce an additional instruction to update the stack pointer.

The lifter bundles VEX instructions in a series of Intermediate Representation Super-Block (IRSB) objects. An IRSB is a single-entry multiple-exit block of consecutive VEX instructions. This definition implies that a VEX interpreter can execute an IRSB only starting from its first instruction until it finds a (possibly conditional) jump to a different IRSB that terminates its execution.

2.2. Symbolic Variables

Symbolic execution, as the name suggests, simulates the execution of a computer program using symbols rather than (or along with) concrete values. Each symbol represents a generic variable that can assume (potentially) any value. Symbolic executors restrict the value of a symbolic variable using logical constraints that usually reflect the series of conditional branches that lead to the path in execution. Symbolic executors use these constraints as the input for Satisfiability Modulo Theories (SMT) solvers that use a combination of logic and heuristics to obtain one or more concrete values for any of the variables involved or prove the unfeasibility of that specific conjunction of constraints.

ByteMatcher employs the version of symbolic variables offered by the `claripy` library in the form of Bit Vector (`BV`) objects. A `BV` represents a variable as a series of bits that can be either concrete if its value is deterministically assigned or symbolic if it is still unknown. `BV` objects allow to address individual or ranges of bits and support arithmetic and logical operations to combine multiple variables into complex formulas in the form of a new `BV`. The library also offers an SMT solver implemented as a `Solver` object. ByteMatcher uses `BV` objects in its memory model to represent the content of memory and registers, as well as memory addresses. Moreover, ByteMatcher exploits the constraint solver to prove the equivalence of any pair of `BV` objects. In particular, two symbolic variables are equivalent if the solver cannot find any possible assignment for which they are different.

$$bv_1 \equiv bv_2 \leftrightarrow \neg Solver(bv_1 \neq bv_2)$$

2.3. Symbolic Loop Summaries

Analyzing loops is one of the most challenging tasks in symbolic execution because they can introduce many execution paths. However, it is also crucial in numerous contexts, like bug detection or test generation.

The authors of [12] propose an approach that generates on the fly (i.e., without the need for static analysis) a logical expression that summarizes the effects of a loop execution.

ByteMatcher applies an approach inspired by their work to handle *symbolic loops*, whose number of iterations is unknown during the symbolic execution. The defining characteristic of a loop of this kind is that the condition deciding whether to continue or stop depends on one or more symbolic variables. The convenience of using a logical expression collides with the fact that, in some instances, it cannot perfectly capture all

the side effects. For this reason, ByteMatcher completely executes *concrete loops* to avoid approximated results as much as possible.

2.3.1. Glossary and Definitions

A loop has two distinct sections with different purposes:

- The *header* is the block of instructions that evaluates the condition controlling the loop execution, called *guard*. This section always ends with a conditional branch.
- The *body* is the set of instructions (one or more blocks) executing at each iteration of the loop. The body is the section whose effects need to be summarized.

Every execution of the header is called a *hit*. During normal execution, a loop always begins and ends with a hit, making the loop's *Execution Count (EC)* the same as the number of hits minus one.

An *Induction Variable (IV)* is a variable of a loop (register or memory address) whose value changes by a constant amount at every execution of the body. IVs are collected in a structure named *Induction Variables Table (IVT)* as tuples of type $T = (B, V, dV, F)$. Table 2.1 describes each of these variables. The final value is computed using the formula $F = B + dV * (EC - 1)$. An *IV candidate* is a loop variable whose status of IV is not yet confirmed nor refuted.

Variable	Description
B	The base value of the IV at the end of the first iteration.
V	The value of the IV computed at the end of each iteration.
dV	The difference in V between two iterations.
F	A symbolic function specifying the final value of the IV.

Table 2.1: Definition of each variable in the IVT tuples.

A guard is a condition of the type $LHS \lhd RHS$, in which $\lhd$ is the *comparison operator*. By defining the guard's value as $D = LHS - RHS$, the condition becomes $D \lhd 0$. Table 2.2 shows the meaning of each variable involved in defining a loop guard. The algorithm can correctly summarize only loops whose guard is *IV-dependent*, meaning that D is an IV. This characteristic is required to accurately estimate the EC of the loop, which is the key to computing the final value of each loop variable.

Variable	Description
LHS	Left-hand side term of the loop guard.
RHS	Right-hand side term of the loop guard.
$\triangleleft$	The comparison operator $\triangleleft \in \{<, >, \leq, \geq, =, \neq\}$.
D	The value of the loop guard updated at every hit.
dD	The difference in D between two iterations.

Table 2.2: Definition of each variable defining a loop guard.

2.3.2. Loop Summary Generation

These are the steps to generate a loop summary:

1. Initialize the loop summary.
2. Collect the IV candidates in the IVT.
3. Update the IVT.
4. Refine the IV candidates.
5. Finalize the loop summary.

Initialization At the first hit of the loop header (i.e., before the first iteration of the body), the algorithm creates an empty IVT and computes the value of the loop guard D for future reference.

Collecting the candidates During the first execution of the loop body, the algorithm creates a tuple for each variable in the IVT initializing the B and V fields, again, for future reference.

Updating the IVT When the algorithm encounters the header for the second time, it computes the new value of the loop guard D and its difference across the executions dD. The algorithm assumes that the loop guard is IV-dependent hence dD will not get any update in the subsequent hits.

Refining the candidates This step takes place at the third and fourth hit. For each of the IV candidates in the IVT, the algorithm computes its new value V along with the difference across the executions dV. If dV changes, the algorithm discards the candidate

for not being an Induction Variable. Repeating this process would technically raise the confidence that each candidate is an IV by refining the content of the IVT even more. Unfortunately, doing this would increase the execution time without guaranteeing higher accuracy, given the assumptions necessary to make the algorithm possible.

Finalization Once the execution of the loop ends, the algorithm estimates the EC using the available data. The exact formula depends on the comparison operand $\lhd$ and which branch stays in the loop. Equation 2.1 shows an example of the estimation of EC for operand $\leq$, assuming that the *else* branch is staying in the loop. By exploiting the EC, the algorithm can compute the final value of each IV in the IVT using the formula already presented. The result is a set of symbolic formulas of type $F = B + dV * (EC - 1)$ that summarize the effects of the loop on each IV depending on the base state B and the number of iterations EC.

$$EC_{\leq,E}(D, dD) = \begin{cases} 0 & \text{if} \quad D \leq 0 \\ \infty & \text{if} \quad D > 0 \land dD \geq 0 \\ (D - dD - 1)/ - dD & \text{if} \quad D > 0 \land dD < 0 \end{cases} \tag{2.1}$$

3 | Approach

3.1. Algorithm Overview

ByteMatcher offers two main functionalities to developers and security experts. The first one allows them to compare the execution of two bytecodes (fragments of compiled programs) to determine whether they are semantically equivalent, meaning whether they are interchangeable for all intents and purposes. The second functionality, if enabled, can detect the presence of (possible) buffer overflow vulnerabilities by analyzing the pattern of memory-related instructions of the bytecode.

Both these functionalities need to analyze code dynamically through symbolic execution. ByteMatcher employs a state-of-the-art symbolic executor to simulate the code. This solution allows focusing on the design of algorithms that customize particular aspects of the symbolic executor's behavior rather than implementing it from scratch.

3.1.1. Symbolic Execution

To correctly implement its functionalities, ByteMatcher submits every bytecode to a step of exploratory analysis before even starting its execution. This exploration constructs the Control Flow Graph (CFG) of the bytecode to extract information on looping paths. In particular, ByteMatcher is interested in each loop's entry and exit addresses and uses this data to guide the symbolic execution step.

The symbolic executor cannot directly work on the bytecode but needs to transform it in an Intermediate Language (IL) through code lifting. The IL provides an abstraction over architecture-specific details, allowing to symbolically execute code originally compiled for any processor in the same way.

Figure 3.1 provides a high-level representation of the division of the tasks inside of ByteMatcher. These three main components interact at different levels during the symbolic execution to obtain the best outcome possible from the code simulation.

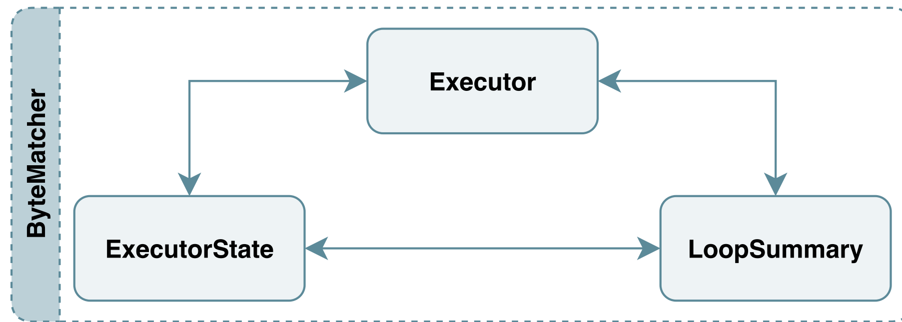

Figure 3.1: High-level overview of ByteMatcher's main components.

Executor This is the entry point of ByteMatcher and the only component interacting with the developers. It receives the bytecode and performs the exploratory analysis on it. It also functions as an orchestrator during the symbolic execution phase. It keeps track of all the execution paths that still need to be explored and decides which one to simulate at each step.

ExecutorState This component represents the machine's state at any point during symbolic execution. It implements a fully-symbolic memory model divided into registers and memory and tracks every operation modifying the machine's state. This component also acts as an interface to the external symbolic executor. Indeed, it implements function handlers that the symbolic executor uses to outsource the simulation of particular instructions. Besides the operations accessing the state, it handles conditional branches and function calls implementing original algorithms.

LoopSummary This component handles a particular case of conditional branches. Input-dependent loops represent one of the leading open challenges of symbolic execution. It is, in fact, notoriously tricky to correctly simulate the body of a loop without knowing the concrete number of iterations. For this reason, this component implements an approach that uses symbolic execution to estimate the number of iterations and produce a reusable logical summary of the loop.

At the end of the symbolic execution, ByteMatcher reconciles all the diverging execution paths in a single `ExecutorState` using a custom approach to merge memory models obtained from the execution of different (related) branches. This unique data structure contains the logical description of the state of every single register and memory address after the execution of the bytecode, parametric in the initial state.

3.1.2. Semantic Equivalence

To test the semantic equivalence of two bytecodes, developers can ask ByteMatcher to compare the results of their execution (in the form of `ExecutorState` objects). Developers can also control the sensitivity of the comparison by selecting one of four equivalence levels that affect the behavior of ByteMatcher. Different equivalence levels might result in different predictions. Because of this, it is crucial to select the right equivalence level for the context of the bytecodes' application.

The algorithm performing the comparison implements an original technique devised during this research. This technique consists in extracting the logical formula describing the final state of (almost) every register and memory address from the first `ExecutorState` and testing it against its counterpart from the second `ExecutorState`. These formulas are compatible if there is no assignment of inputs in the initial state (common to both executions) that produces a different concrete value for that register or memory address. Two bytecodes are semantically equivalent only if all tests produce a positive result.

3.1.3. Buffer Overflow Detection

Developers can instruct ByteMatcher to report the detection of (possible) buffer overflow vulnerability during the symbolic execution phase. ByteMatcher provides this function through another original technique that analyzes all the operations that change the content of the memory during the execution of a symbolic loop. The loop summarization algorithm takes note of all those instructions that update the content of the memory based on a symbolic pointer. Whenever ByteMatcher uses a loop summary, if it cannot concretize the address of the target of a write operation, it flags it as odd behavior, triggering the action to report this possible vulnerability to the developer. This technique works on the reasonable assumption that write operations with symbolic pointers are dangerous, especially in the context of symbolic loops. Because of this, detecting an overflow does not guarantee that the bytecode is vulnerable also in real scenarios.

3.2. Approach Details

3.2.1. Fully-Symbolic Memory Model

It is possible to summarize the state of a program during its execution by describing the content of the registers and the allocated memory (stack and heap). For this reason, to correctly simulate a program, it is essential to define a structure (*memory model*) that

stores such information together with a protocol describing how to interact with it.

The memory model for ByteMatcher is inspired by the description of MemSight [4, 8], an approach to symbolic execution that offers support for memory operations on symbolic addresses by providing a compact representation without the need of concretization. ByteMatcher splits the state between registers and memory, each implementing MemSight differently. Both registers and memory support three fundamental operations: read, write, and merge. While the first two are straightforward, the merge operation enables the option of joining the content of two data structures and plays a central role in the ByteMatcher's approach as outlined in Chapter 4.

Registers

ByteMatcher describes the state of the processor registers as a set of tuples of the form $T = (r, v, \tau, \delta)$. Table 3.1 describes each of these variables. The size of the symbolic value v matches the number of bits of the code's architecture. An object of type `RegisterState` stores the tuples along with two counters, one that increases by one at every write operation (τ_+) and one that decreases by the same amount at every read operation (τ_-).

Variable	Description
r	Unique identifier of the register.
v	Concrete or symbolic value assigned to the register.
τ	Insertion time of the tuple.
δ	Logical condition under which the tuple is valid.

Table 3.1: Definition of each variable in the register tuples.

Write operation Algorithm 3.1 shows the pseudo-code for the function that writes the value of a register. The statement at Line 3 increases the time variable as anticipated. After that, the algorithm standardizes the size of the value to be written v by either extending it with the content already present in the register (Line 4) or by clipping it to the desired size (Line 7). Finally, the algorithm instantiates a new register tuple and adds it to the set of tracked tuples (Lines 10-11).

Read operation Algorithm 3.2 shows the pseudo-code for the function that reads the value of a register. The read operation retrieves the value of a register at any time $\hat{\tau}$. The algorithm begins by filtering all the tuples related to the desired register and sorting them

Algorithm 3.1 Register write operation.

```
1: global τ+, tuples, arch
2: function RegisterWrite(r, v)
3:     τ+ ← τ+ + 1
4:     if v.size < arch.bits then
5:         r_pre ← RegisterRead(r, arch.bits, τ+)
6:         v ← v ⊕ r_pre[v.size : arch.bits − 1]    ▷ ⊕ represents the concatenation operator
7:     else if v.size > arch.bits then
8:         v ← v[0 : arch.bits − 1]
9:     end if
10:    tuple ← new RegisterTuple(r, v, τ+, true)
11:    tuples ← tuples ∪ tuple
12: end function
```

by increasing time (Lines 3-4). If no tuple exists, the algorithm injects a symbolic value as a placeholder at the negative time τ_- to symbolize an uninitialized read (Line 6-10). Starting at Line 11, a loop over the tuples produces an expression made of nested if-then-else (ITE) operators. Traversing the tuples in chronological insertion order ensures that the resulting expression evaluates the most recent tuple before anything else. Finally, the resulting expression is sliced to the desired size and returned (Line 17).

Algorithm 3.2 Register read operation.

```
1: global τ−, tuples
2: function RegisterRead(r, size, τ̂)
3:     filtered ← filterByRegister(tuples, r)
4:     sorted ← sortByTime(filtered)
5:     v ← uninit(r)                            ▷ Symbolic value that depends on r
6:     if sorted.isEmpty then
7:         τ− ← τ− − 1
8:         tuple ← new RegisterTuple(r, v, τ−, true)
9:         tuples ← tuples ∪ tuple
10:    end if
11:    for tuple in sorted do
12:        if tuple.τ > τ̂ then
13:            break
14:        end if
15:        v ← ite(tuple.δ, tuple.v, v)
16:    end for
17:    return v[0 : size − 1]
18: end function
```

Merge operation Algorithm 3.3 shows the pseudo-code for the function that merges two `RegisterState` objects R_1 and R_2 by updating their closest common ancestor A.

Each of the two `RegisterState` is associated with a logical condition under which its content is valid (c_1 and c_2, respectively). The algorithm executes two loops. The first, extracts all the tuples that are in R_1 but not in the ancestor A (Line 3) and injects them in A adding c_1 to the tuple's condition δ (Line 4). The second, repeats the same operation for R_2. At Lines 9-10, it updates the ancestor times to preserve a consistent state.

Algorithm 3.3 Register merge operation.

```
1: function REGISTERMERGE(R₁, c₁, R₂, c₂, A)
2:     for tuple in R₁.tuples do
3:         if tuple.τ > A.τ₊ ∨ tuple.τ < A.τ₋ then
4:             tuple.δ ← tuple.δ ∧ c₁
5:             A.tuples ← A.tuples ∪ tuple
6:         end if
7:     end for
8:     ...                                  ▷ Repeat Lines 2-7 for R₂
9:     A.τ₊ ← MAX(R₁.τ₊, R₂.τ₊)
10:    A.τ₋ ← MIN(R₁.τ₋, R₂.τ₋)
11: end function
```

Memory

Similarly to registers, BYTEMATCHER describes the state of the memory as a set of tuples. There are two different kinds of tuples, one generated by operations on concrete addresses $T_c = (a_c, v, \tau, \delta)$ and one by operations on symbolic addresses $T_s = (a_{min}, a_{max}, a_s, v, \tau, \delta)$ (Table 3.2). An object of type `MemoryState` stores the two sets of concrete and symbolic tuples along with the τ_+ and τ_- counters already presented for registers.

Variable	Description
a_c	Concrete memory address.
a_s	Symbolic memory address.
a_{min}	The smallest concrete value that a_s can assume.
a_{max}	The largest concrete value that a_s can assume.
v	Concrete or symbolic value assigned to the byte of memory.
τ	Insertion time of the tuple.
δ	Logical condition under which the tuple is valid.

Table 3.2: Definition of each variable in the memory tuples.

Write operation Algorithm 3.4 shows the pseudo-code for the function that writes a value of arbitrary size starting from a memory address. Since ByteMatcher handles the memory in bytes, the first step of the write operation is to chop the value v into single byte values (Line 2), which then calls Algorithm 3.5 that writes it in consecutive addresses starting from a. Depending on the nature of the address, the second algorithm creates and stores either a concrete tuple (Line 6) or a symbolic tuple (Line 9).

Algorithm 3.4 Multi-byte memory write operation.

```
1:  function MemoryWrite(a, v)
2:      bytes ← CHOP(v)
3:      for byte in bytes do
4:          MemoryWriteByte(a, byte)
5:          a ← a + 1
6:      end for
7:  end function
```

Algorithm 3.5 Single-byte memory write operation.

```
1:  global τ+, tuples_c, tuples_s
2:  function MemoryWriteByte(a, v)
3:      a_min ← MIN(a)
4:      a_max ← MAX(a)
5:      τ+ ← τ+ + 1
6:      if a_min = a_max then
7:          tuple ← new ConcreteTuple(a_min, v, τ+, true)
8:          tuples_c ← tuples_c ∪ tuple
9:      else
10:         tuple ← new SymbolicTuple(a_min, a_max, a, v, τ+, true)
11:         tuples_s ← tuples_s ∪ tuple
12:     end if
13: end function
```

Read operation Algorithm 3.6 shows the pseudo-code for the function that reads an arbitrary number of bytes from memory. As for the write operation, ByteMatcher can read consecutive memory addresses in one operation by loading each byte using Algorithm 3.7 and concatenating it in a single symbolic value (Line 6). The procedure to read a byte from memory is similar to the one for registers, the main difference being the function beginning at Line 21. The search operation extracts all the tuples whose address overlaps the given range (between a_{min} and a_{max}) and whose insertion time precedes the given time $\hat{\tau}$.

Algorithm 3.6 Multi-byte memory read operation.

```
1: function MemoryRead(a, size, τ̂)
2:     bytes ← ∅
3:     for i ← 0; i < size; i ← i + 1 do
4:         bytes ← bytes ∪ MemoryReadByte(a + i, τ̂)
5:     end for
6:     return concat(bytes)
7: end function
```

Algorithm 3.7 Single-byte memory read operation.

```
 1: global τ_−, tuples_c, tuples_s
 2: function MemoryReadByte(a, τ̂)
 3:     a_min ← min(a)
 4:     a_max ← max(a)
 5:     valid ← MemorySearch(a_min, a_max, τ̂)
 6:     sorted ← sortByTime(valid)
 7:     v ← uninit(a)
 8:     τ_− ← τ_− − 1
 9:     if a_min = a_max then
10:         tuple ← new ConcreteTuple(a_min, v, τ_−, true)
11:         tuples_c ← tuples_c ∪ tuple
12:     else
13:         tuple ← new SymbolicTuple(a_min, a_max, a, v, τ_−, true)
14:         tuples_s ← tuples_s ∪ tuple
15:     end if
16:     for tuple in sorted do
17:         v ← ite(tuple.a = a ∧ tuple.δ, tuple.v, v)      ▷ tuple.a can be either a_c or a_s
18:     end for
19:     return v
20: end function
21: function MemorySearch(a_min, a_max, τ̂)
22:     valid ← ∅
23:     for tuple in tuples_c do
24:         if a_min ≤ tuple.a ≤ a_max ∧ tuple.τ ≤ τ̂ then
25:             valid ← valid ∪ tuple
26:         end if
27:     end for
28:     ...                                              ▷ Repeat Lines 23-27 for tuples_s
29:     return valid
30: end function
```

Merge operation Algorithm 3.3 shows the pseudo-code for the function that merges the content of two `MemoryState` objects M_1 and M_2. The algorithm is similar to the one presented for registers.

Algorithm 3.8 Memory merge operation.

```
1: function MemoryMerge(M_1, c_1, M_2, c_2, A)
2:     for tuple in M_1.tuples_c do
3:         if tuple.τ > A.τ_+ ∨ tuple.τ < A.τ_- then
4:             tuple.δ ← tuple.δ ∧ c_1
5:             A.tuples_c ← A.tuples_c ∪ tuple
6:         end if
7:     end for
8:     ...                  ▷ Repeat Lines 2-7 for M_1.tuples_s, M_2.tuples_c, and M_2.tuples_s
9:     A.τ_+ ← max(M_1.τ_+, M_2.τ_+)
10:    A.τ_- ← min(M_1.τ_-, M_2.τ_-)
11: end function
```

3.2.2. Uninterpreted Function Calls

In mathematical logic, the term uninterpreted function describes the practice of representing a function only through its signature (name and arguments). The uninterpreted function theory describes complex functions whose effects are unknown or uninteresting, depending on the context, via a symbol.

ByteMatcher leverages the uninterpreted functions theory to handle any call during the symbolic execution of a bytecode. This strategy simplifies the simulation by skipping parts of the code altogether. In particular, calls are a critical topic because:

- The function might have some unwanted side effects that ByteMatcher cannot track (e.g., system or user interaction).
- The function might be out of the scope of the bytecode, thus being unknown to ByteMatcher (e.g., library functions).
- Even if the previous points do not apply, a call disrupts the execution flow and increases the overall complexity of the simulation.

Whenever ByteMatcher encounters a call, it uses the function's name and arguments to produce a so-called fingerprint, a symbolic value that depends only on the function's signature. ByteMatcher uses that fingerprint as the function's return value, allowing the simulation to continue without issue. By applying this behavior, ByteMatcher can correctly match two bytecodes invoking the same function without the extra burden of simulating it.

3.2.3. Comparison and Equivalence Levels

Compare Operation

ByteMatcher considers computer programs as black boxes that it simulates without knowledge or interest in the specific instructions. The focus of ByteMatcher is only on the input and output states. From this point of view a program P is just a block that transforms a series of inputs $i_1, i_2, ..., i_n$ into a series of outputs $o_1, o_2, ..., o_m$ (Figure 3.2).

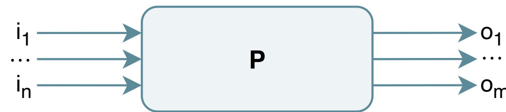

Figure 3.2: Representation of a program as a black box.

On real machines, inputs and outputs are stored either in registers or in memory. These two locations constitute the state of that machine. Any generic program can access the whole state of the input machine M_i, transforming it into a new state M_o during the execution (Figure 3.3).

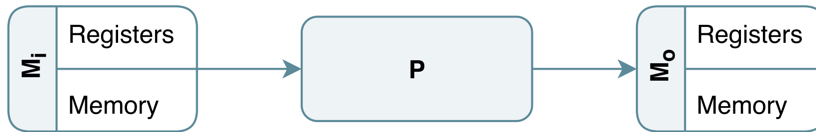

Figure 3.3: Definition of the machine's state in the black box representation.

From this point of view, two programs P_1 and P_2 are semantically equivalent if, given the same input state M_i, their respective output state M_o^1 and M_o^2 are equivalent, or, more in practice, if their codes produce the same effect on the state of a machine and are therefore interchangeable from the point of view of a black box (Figure 3.4).

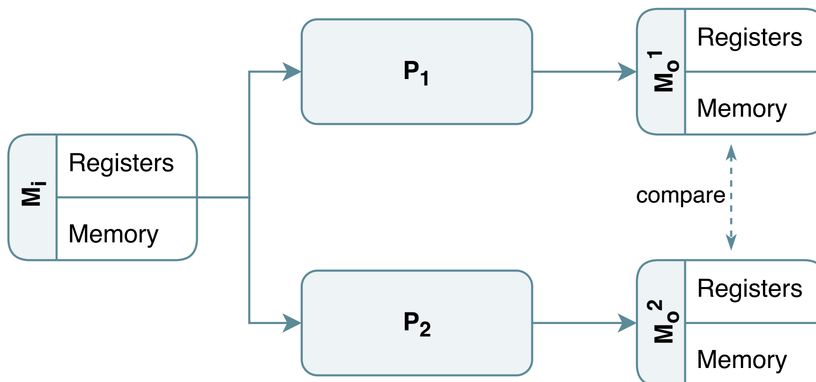

Figure 3.4: Semantic equivalence in the black box representation.

Equivalence Levels

When comparing the execution of two bytecodes, ByteMatcher offers to the developers a series of predefined settings to control its sensitivity altering the behavior of the comparison algorithm and, therefore, the result of the procedure. In detail, ByteMatcher provides the following levels of equivalence:

- `Lite`: skips the comparison of pointer registers (IP, SP, BP) and other artificial registers (e.g., temporary registers). `Lite` is the default level of equivalence.
- `API`: skips the comparison of all the registers saved by the caller according to the architecture's calling convention. ByteMatcher implements it for the x86_64 architecture as a proof of concept.
- `Memory`: ignores all registers during the comparison operation.
- `Total`: compares the entire content of both registers and memory.
- `Permutation`: compares the overall content of registers and memories even when the two codes use different temporary registers or variables are allocated in mixed order. ByteMatcher does not currently implement this level of equivalence.

In addition, ByteMatcher allows developers to provide the comparison algorithm a list of register names (according to the specific architecture of the bytecode) to ignore regardless of the level of equivalence selected.

3.2.4. Buffer Overflow Detection

Developers can instruct ByteMatcher to report the detection of (possible) buffer overflow vulnerabilities. In particular, the tool analyzes memory operations during the symbolic execution of the bytecode, looking for unsafe patterns that might indicate a vulnerability. This way of detecting vulnerabilities in the code has roots in the reasonable assumption that every register or memory address that is uninitialized at the beginning of the execution of the bytecode might be part of its inputs. Since it is a good security practice never to trust user input, ByteMatcher considers uninitialized variables potentially unsafe.

While symbolic write operations are not problematic in the generic case, they might become a security risk if they happen inside a symbolic loop. A loop is symbolic if the number of iterations of its body is unknown, meaning that it depends on some combination of uninitialized variables. On the other hand, a symbolic write happens when the target address is unknown and, as said, potentially controllable. Putting these two things

together means the user can control where and how many times to write, thus the buffer overflow vulnerability.

The vulnerability detection happens in the algorithm that applies a `LoopSummary` to an `ExecutorState`. Whenever ByteMatcher cannot concretize the address of a symbolic memory write using the context provided by the `ExecutorState`, that operation is considered malicious and reported to the developer as potentially vulnerable. Nevertheless, since ByteMatcher analyzes out-of-context bytecodes and not entire binaries, this technique to detect buffer overflow vulnerabilities does not guarantee the existence of a real threat.

4 | Implementation Details

4.1. System Architecture

This Section provides an overview of the structure of the ByteMatcher project and the technologies contributing to its development.

4.1.1. Tools and Technologies

ByteMatcher is implemented using Python, the de-facto standard in computer security and binary exploration. Python is the perfect programming language to build a project quickly and reliably. It is open-source and offers an extensive collection of external libraries to support developers in any aspect of the development process, from debugging to performance optimization.

At the core of ByteMatcher, there is Angr [21], whose team offers an open-source library implementing their framework for symbolic execution. In particular, among the ones packaged with Angr, ByteMatcher makes use of the following libraries:

- `archinfo`: it exposes a collection of classes containing architecture-specific information such as the number of bits in a word, the list of registers, or the endianness of the memory.
- `pyvex`: originally developed for [20], it is a Python interface for VEX intermediate language that is essential to symbolically execute a bytecode as explained in Section 2.1.
- `claripy`: it provides an interface for `Z3` [11], a powerful Satisfiability Modulo Theories (SMT) solver from Microsoft.
- `networkx`: a library for the creation and analysis of complex data structures, which is at the center of Angr's Control Flow Graph (CFG) implementation.

4.1.2. Project Structure

Figure 4.1 shows the structure of ByteMatcher, grouping classes of objects by their main functionality:

- Controller objects handle the symbolic execution and most business logic.
- Model objects implement the memory model described in Subsection 3.2.1.
- Data objects provide an interface for the tuples employed in the memory model.

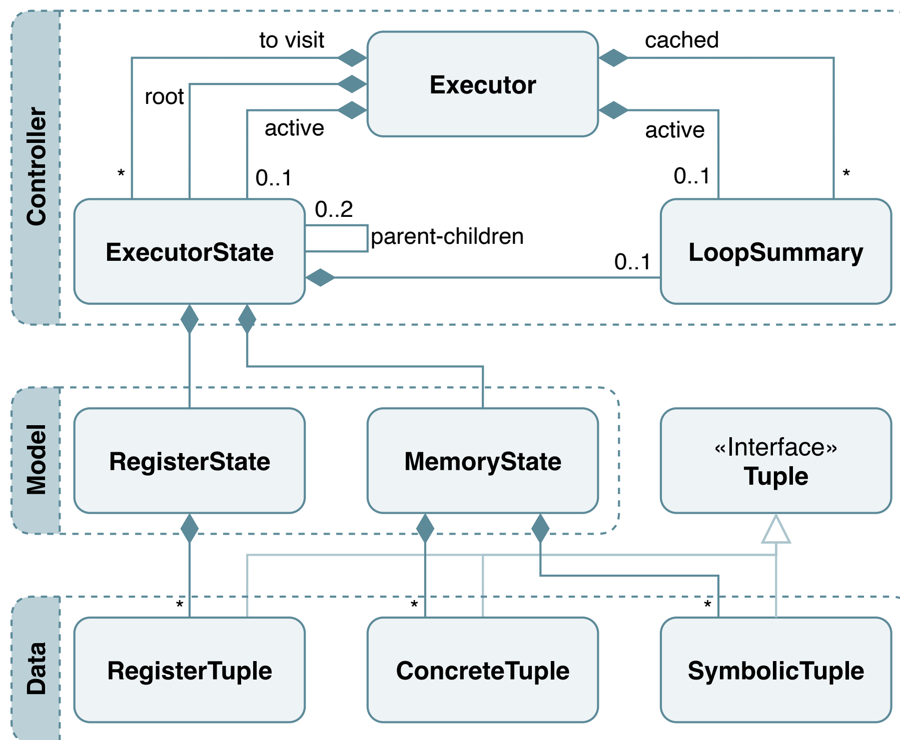

Figure 4.1: UML class diagram of ByteMatcher.

Controller

Executor It is the only point of contact with the external world. It serves as an orchestrator for the symbolic execution and stores the simulation's results. Moreover, developers can compare multiple `Executor` objects to test for equality. To track the state of the simulation process, `Executor` builds a tree of objects of type `ExecutorState`. In particular, during symbolic execution, it keeps a reference to the tree's root and the active state under analysis, along with a queue of `ExecutorState` objects waiting to be visited by the algorithm. At the end of the simulation, the queue is empty, there is no active state, and a single `ExecutorState` represents the collapsed state tree, referenced by the root. When dealing with symbolic loops, `Executor` objects invoke a new instance of their same class

to perform the summarization task in a detached environment. In this case, the inner `Executor` receives an object of type `LoopSummary` containing information about the loop. `LoopSummary` tracks any update during the summarization process. The outer `Executor` caches the `LoopSummary` object and applies it instead of executing the symbolic loop.

ExecutorState This object behaves as the node of a bidirectional tree, meaning that every object holds a reference to its parent node and (up to) two children. Moreover, `ExecutorState` implements the memory model in the form of a couple of `RegisterState` and `MemoryState` objects. `ExecutorState` exposes methods that are used by the `pyvex` library during the execution of IRSBs. Besides the methods to read and write from registers and memory, it provides handlers to manage function calls and conditional branches. The latter plays an essential role in the simulation as it generates new `ExecutorState` objects that act as children of the current tree node. The branch handler is also crucial to identify symbolic loops, and it signals the execution of the loop header.

LoopSummary As mentioned before, this object holds specific information about a symbolic loop, such as the addresses of its header and body sections or its exit target. `LoopSummary` implements the algorithm of Section 2.3 by providing support methods for `ExecutorState` objects. Within each `LoopSummary` object (Figure 4.2), there is an `IVTable` that tracks the IV candidates during the summarization process as tuples of type `IVTableEntry`. Once the process terminates, the `IVTable` takes the final value of each IV represented by a logical formula in the form of a `BV` and decomposes it in variables that are organized in a `DependencyGraph` to make the process of applying the loop summary to an `ExecutorState` easier.

Model and Data

`RegisterState` and `MemoryState` objects have an essentially similar structure as they both implement the read, write and merge operations presented in Subsection 3.2.1 to manipulate the content of registers and memory, respectively. Special data objects of type `Tuple` store the content of the machine, and their actual implementation changes based on the type of data and the operation used to create it. `RegisterState` only manages objects of type `RegisterTuple`, while `MemoryState` stores the data in `SymbolicTuple` or `ConcreteTuple` objects depending on whether the memory address is symbolic. Both `RegisterState` and `MemoryState` also implement the compare operation that tests the content of two objects of the same type for equivalence.

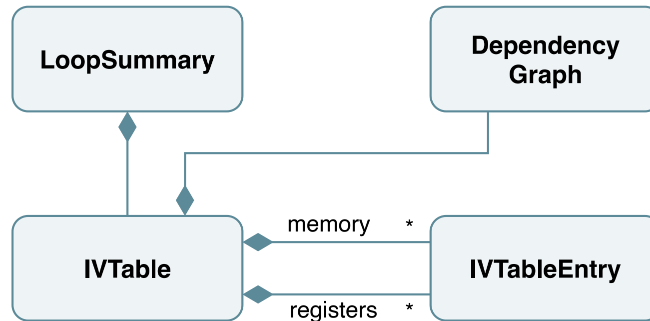

Figure 4.2: Detail of the UML class diagram for `LoopSummary`.

4.2. System Details

This Section dives into the details of implementing the most complex classes presented in Section 4.1. It also maps the approach of Chapter 3 with the code of ByteMatcher.

4.2.1. Executor

The user can symbolically execute a bytecode by instantiating an `Executor`. This process only requires the following information:

- The string of bytes representing the instructions to simulate.
- The processor architecture for which the code is compiled (from `archinfo`).
- The (optional) address of the first instruction in memory.

Exploratory Analysis and Initialization

At the beginning of its initialization routine, `Executor` uses some of the tools offered by Angr [21] to analyze the bytecode. This pre-processing step is essential to gather data, like the flow of execution and the number and type of loops used to guide the simulation.

The method `load_shellcode`, inherited from Angr, transforms the user input into an object of type `Project`. The `Project` class is the main entry point of Angr. It contains a set of binaries and the relationships among them and exposes a variety of analyses to gain all sorts of knowledge about those binaries.

To unlock the full potential of Angr, `Executor` needs to build the Control Flow Graph (CFG) of the bytecode. It does so through the `CFGFast` analysis that quickly produces a graph of nodes of type `CFGNode` using a heuristic approach.

A CFG is a directed graph highlighting all the paths the code might traverse during its execution. Every node represents a basic block of consecutive instructions that starts with a jump target (or the entry point) and ends with a jump. There are no other jumps or jump targets in a block. The edge from node A to node B exists only if B is among the targets of the jump in A.

The result of `CFGFast` might present cycles. For this reason, `Executor` exploits another functionality of ANGR to collect data on those loops when they exist. The `LoopFinder` analysis gathers all this information in a set of `Loop` objects.

Finally, `Executor` performs operations to initialize the data structures employed during the symbolic execution:

- It stores the `Loop` objects in a dictionary indexed by the address of their header. This dictionary also serves as a cache for the generated `LoopSummary` objects.
- It prepares an empty dictionary to store the fingerprints computed for function calls indexed by the address of the call instruction.
- It creates a blank `ExecutorState` that becomes both the tree's root and the active state. The queue of states waiting to be visited is empty instead.

Symbolic Execution

The symbolic execution of the bytecode begins immediately upon initialization, without the need for additional user interaction. This choice bypasses the possibility of having `Executor` objects in incoherent states.

For the symbolic execution, `Executor` assumes that the part of memory containing executable code is only the one occupied by the bytecode, that is, the block of n consecutive bytes beginning at address a where n is the size of the string of bytes and a is either the address provided by the user or a constant default value. Any attempt to lift any code outside this area shall fail.

`Executor` simulates the code inside its main loop that breaks if and only if, after an `ExecutorState` is deactivated, it is not able to find a new state to activate (i.e., the queue of states to visit is empty).

At each iteration of the main loop, `Executor` queries the active state for the value of the Instruction Pointer (IP) register, which contains the address of the next instruction to be executed (*base address*). At this point, `Executor` exploits the `pyvex` library to lift as many instructions as possible, starting from the base address. If the process of lifting

fails, the execution of the current state ends, and a new state is activated.

After obtaining a valid IRSB, `Executor` checks whether the cache contains a `LoopSummary` for the base address. In such a case, `Executor` applies it to the active state, and the execution skips to the next iteration of the main loop. Subsection 4.2.3 discusses the details of the loop summarization.

Otherwise, `Executor` instructs the active `ExecutorState` to symbolically execute the IRSB as illustrated in Subsection 4.2.2. If `ExecutorState` detects a symbolic loop during the simulation, `Executor` triggers the specific routine to generate and apply a `Loop-Summary`.

After the analysis, `Executor` appends the newly-created children of the active state to the queue and selects a new `ExecutorState` to activate for the following iteration.

Consolidation of the State Tree

`Executor` executes this step at the end of its initialization routine. It consists of collapsing the tree of `ExecutorState` objects built during the code simulation in a single state whose memory model accounts for all the possible paths taken during the execution. The root attribute stores the super-state of the tree that is the object involved when comparing one `Executor` to another to test for equivalence.

`Executor` traverses the state tree using a DFS strategy and collapses every node N according to the following rules:

1. If N has no children (i.e., is a leaf), then N is already the super-state.
2. If N has one child C, then C is the super-state for the tree rooted in N.
3. If N has two children $C1$ and $C2$, then the super-state for the tree rooted in N is the result of the merge between $C1$ and $C2$.

4.2.2. ExecutorState, RegisterState, and MemoryState

`ExecutorState` is a crucial component for symbolic execution. It inherits from the class `VEXMixin` of `pyvex` that exposes a single interface through which the symbolic executor can modify the state of the hypothetical machine running the code. `VEXMixin` provides, among others, the `handle_vex_block` method that `Executor` needs to simulate a IRSB. During the symbolic execution, `VEXMixin` triggers several handlers, each of which mimics the effect of a specific type of instruction. Following the mixin pattern from Python, `ExecutorState` overrides some of these handlers to implement custom functions.

The `Get`/`Put` and the `Load`/`Store` functions (of type `_perform_vex_expr_`) handle instructions that interact with registers and memory respectively. `ExecutorState` forwards these requests to the `RegisterState` and `MemoryState` objects that fulfill them as illustrated in Subsection 3.2.1.

The `_perform_vex_expr_CCall` reacts by definition to calls to pure (without side-effects) C helper functions and takes as arguments the name of the function and the list of arguments. The version implemented in `ExecutorState` computes the fingerprint of the function call as explained in Subsection 3.2.2 and uses it as the return value. However, `Executor` manually invokes this handler whenever a IRSB ends with an anonymous call to a memory address. In this case, the name of the function is replaced by the string "anonymous" while the arguments are retrieved using the following heuristics:

1. Reverse the instructions of the IRSB
2. According to the architecture, record all the writes to argument registers.
3. Stop at the first write to a non-argument register.

Finally, `pyvex` calls the `_perform_vex_stmt_Exit` function whenever the IRSB ends with a conditional branch to select the jump target. This handler plays a crucial role in ByteMatcher and manages various situations. Algorithm 4.1 illustrates the function. The boolean variables computed at Lines 12-13 ensure to append to the state tree only the `ExecutorState` corresponding to feasible jump targets. When present, `LoopSummary` handles the loop summarization. In such case, the behavior of `ExecutorState` mutates to keep only one target for each loop iteration (Lines 16-22). Like the handler analyzed before, `Executor` also calls the `Exit` function during its main loop on two special occasions:

- The IRSB ends with an unconditional jump.
- The IRSB ends with an anonymous function call.

In both cases, `Executor` needs the handler to correctly spawn a child `ExecutorState` and continue the simulation.

Merge Operation

As seen in Subsection 4.2.1, during the lifetime of `Executor`, the tree of states needs to be collapsed in one super-state, possibly requiring merging two `ExecutorState` objects. The procedure to merge two states consists in joining the content of their `RegisterState` and `MemoryState` objects through the merge operation illustrated in Subsection 3.2.1.

Algorithm 4.1 Handler for conditional branches in `ExecutorState`.

```
1: global irsb, constr, summary, max_hits ← 4
2: function EXIT(guard, target)
3:     target_jt ← target                              ▷ jt stands for jump taken
4:     target_jnt ← irsb.default_exit_target           ▷ jnt stands for jump not taken
5:     g ← guard ≠ 0
6:     consta_jt ← constr ∪ g
7:     consta_jnt ← constr ∪ ¬ g
8:     gdt ← ISTRUE(g)                                 ▷ Guard definitely true
9:     gdf ← ISFALSE(g)                                ▷ Guard definitely false
10:    gpt ← ISSATISFIABLE(consta_jt)                  ▷ Guard possibly true
11:    gpf ← ISSATISFIABLE(consta_jnt)                 ▷ Guard possibly false
12:    jt ← gdt ∨ (¬ gdf ∧ gpt)
13:    jnt ← gdf ∨ (¬ gdt ∧ gpf)
14:    if summary then
15:        summary.HANDLEHIT(g)                        ▷ Algorithm 4.2
16:        if summary.hit < max_hits then              ▷ Keep only the looping branch
17:            jt ← summary.jt_in_loop
18:            jnt ← summary.jnt_in_loop
19:        else                                        ▷ Keep only the breaking branch
20:            jt ← ¬ summary.jt_in_loop
21:            jnt ← ¬ summary.jnt_in_loop
22:        end if
23:    end if
24:    if jt then
25:        child_jt ← CLONE(consta_jt, target_jt)
26:    end if
27:    if jnt then
28:        child_jnt ← CLONE(consta_jnt, target_jnt)
29:    end if
30: end function
```

Compare Operation

Following the theoretical representation of a computer program as a black box introduced in Subsection 3.2.3, `ExecutorState` objects represent the machine's state, and the bytecode simulated by `Executor` represents the program (Figure 4.3). The input of the symbolic execution is a fully-symbolic `ExecutorState` in which (almost) every register and memory address is uninitialized, and the output is a different `ExecutorState` reflecting the effects of the computation. Therefore, two bytecodes are semantically equivalent if their output `ExecutorState` objects are equivalent. The outputs are the super-states of the respective `Executor` objects.

Figure 4.3: Black box representation in ByteMatcher.

The comparison of two `ExecutorState` objects is "point-to-point" in the sense that it tests the value of each register or memory address in the first state only to its analogous in the second state. The value of each entry is a `BV` extracted performing a read on the `RegisterState` or `MemoryState`. The comparison uses `claripy`'s constraint solver to define whether the two entries match (refer to Section 2.2 for more information on `BV` objects). To improve the performance of the process, ByteMatcher restricts the comparison only to entries `in use`. A register or a memory address is in use if `RegisterState` and `MemoryState` contain at least one `Tuple` modifying its value. As anticipated in Subsection 3.2.3, depending on the `EquivalenceLevel` of choice and the user's input, some entries might be ignored during the comparison process.

4.2.3. LoopSummary

`LoopSummary` serves a triple function:

- It provides information about a symbolic loop (the `Loop` object from Angr) employed during the execution.
- It stores the data required for the summarization and orchestrates the algorithm to produce the summary.
- It offers a method used by `Executor` to apply the results of a symbolic loop to any `ExecutorState` object.

Section 2.3 presents in-depth theoretical concepts and a high-level description of the algorithm for loop summarization.

Symbolic Loop Detection

The detection of a symbolic loop happens during the main loop of the simulation in `Executor`. Thanks to the exploratory analysis performed using Angr, `Executor` already has a dictionary of all the loops in the CFG of the bytecode. When the address of the IRSB in execution matches the address of the header of a loop, if the active `Executor-`

State ends with two children (i.e., the jump condition is symbolic), Executor triggers the generation of a LoopSummary.

After the summary is complete, Executor caches it and rewinds the execution, restarting the execution from the parent of the active ExecutorState. Executor applies the summary the next time the loop header becomes active.

Loop Summarization Algorithm

The outer Executor kicks off the summarization process by initializing a LoopSummary with the data about the loop and by passing it to a new (inner) Executor. The inner Executor can simulate only the blocks of code specified by the Loop object in a new isolated environment.

During the symbolic execution of the first iteration of the loop body, every write operation triggers a side effect that enters a new IVTableEntry to the IVTable as a candidate.

When the branch handler of ExecutorState is triggered by the header of the loop (hit), LoopSummary runs Algorithm 4.2. The algorithm's behavior changes slightly based on the number of iterations already simulated. The first two times (Lines 5-9), LoopSummary extracts from the loop guard the information needed to estimate the Execution Count (EC). The rest of the hits (Lines 10-11), update the IVTable as shown in Algorithm 4.3.

Algorithm 4.2 Handler for a loop header hit.

```
1:  global hit, op, D, dD, ivt
2:  function HandleHit(g)
3:      hit ← hit + 1
4:      lhs, rhs ← g.args
5:      if D is None then                         ▷ Hit #1
6:          op ← g.op
7:          D ← lhs − rhs
8:      else if dD is None then                   ▷ Hit #2
9:          dD ← D − (lhs − rhs)
10:     else                                      ▷ Hits #3+
11:         UpdateIVT()
12:     end if
13: end function
```

As soon as the simulation in the inner Executor finishes, the LoopSummary is finalized by estimating the EC from the data collected during the execution and by using it to compute the final value of each IVTableEntry that is still in the IVTable. During this process, LoopSummary builds a DependencyGraph that recursively connects each BV to its

arguments. The recursion stops whenever it cannot further decompose a `BV`. In such case, the `BV` is a graph entrance.

Extracting the arguments is usually made possible by the `args` property of `BV` objects that returns a list of objects appearing in the formula of the `BV`. Unfortunately, this property is empty for values generated in response to an uninitialized read that lose the dependency from the register or the memory address they represent.

Algorithm 4.3 Routine to update the `IVTable`.

```
1:  global ivt
2:  function UpdateIVT
3:      for idx, entry in ivt do
4:          if entry.type = "memory" then
5:              V ← MemoryRead(idx)
6:          else
7:              V ← RegisterRead(idx)
8:          end if
9:          dV ← V − entry.V
10:         entry.V ← V
11:         if entry.dV is None then
12:             entry.dV ← dV
13:         else if entry.dV ≠ dV then
14:             ivt.remove(idx)
15:         end if
16:     end for
17: end function
```

To recover from this situation, `LoopSummary` checks the dependencies stored in the annotations of the `BV` every time the `args` property is empty while building the `DependencyGraph`. Since the loop execution happens in a fully-symbolic environment, each graph entrance is guaranteed to have annotations. The library `claripy` provides a mechanism to annotate `BV` objects with instances of classes inheriting from `Annotation`. Therefore, when `RegisterState` or `MemoryState` objects generate a new uninitialized value, they annotate it with additional information to help the summarization process (Table 4.1).

Application of the Loop Summary

The first step for applying a `LoopSummary` to an `ExecutorState` is to recursively generate a dictionary with the translations from the fully-symbolic world to the scope of the `ExecutorState` by traversing the `DependencyGraph`.

For each entrance of the graph, `LoopSummary` reads its value from the `RegisterState` or `MemoryState` of the state depending on its `TypeAnnotation` and adds it to the dictionary.

	Annotation	Description
Reg	TypeAnnotation	States that the BV belongs to a register.
	ConcreteDep.Ann.	Identifier of the register.
Mem	TypeAnnotation	States that the BV belongs to the memory.
	ConcreteDep.Ann.	Concrete memory address (for concrete reads).
	SymbolicDep.Ann.	Symbolic memory address (for symbolic reads).

Table 4.1: Annotations of uninitialized BV objects.

After that, the algorithm traverses the DependencyGraph with a BFS strategy starting from the entrance and updates the value of each node using the replace_dict method of BV. The resulting dictionary tracks the translation of all the visited nodes.

To apply the effects of the loop on the ExecutorState, LoopSummary uses the dictionary to compute the final value F of every IVTableEntry in the IVTable and performs a write to the RegisterState or MemoryState. In the case of a memory entry, LoopSummary also translates its address before the write.

Finally, the algorithm updates the IP register with the address of the first instruction after the loop. After this operation, the ExecutorState represents the state of the machine after the complete execution of the loop, and the simulation can continue in the outer Executor with the next ExecutorState.

4.2.4. Additional Features

This Subsection presents some additional features implemented to help during the development of ByteMatcher that enhance the usability and are interesting from the point of view of the overall complexity.

Initial Configuration

The user can customize the behavior of ByteMatcher by providing key-value settings through the config.yml file. The following list presents the available options along with their default value.

- stop_first_mismatch (true): if true, the comparison algorithm stops at the first register or memory address that is not equivalent in the two Executor objects. Otherwise, the comparison continues logging all the mismatches. The return value of the comparison function is not affected in any way.

- `stop_buffer_overflow` (`false`): if true, applying a loop summary might throw an exception of type `BuffOverflowException` to report a (possible) buffer overflow vulnerability, as seen in Subsection 3.2.4.
- `remove_register_overlap` (`false`): if true, `RegisterState` deletes tuples with the same register identifier and δ condition on write operations. This option did not provide any real speedup during the execution.
- `concrete_stack_addresses` (`true`): if true, `Executor` initializes SP and BP registers with a concrete value. This option greatly improves the speed of the execution because, usually, memory operations are relative to one of these registers, and symbolic references sensibly increase the complexity of the `BV` objects.
- `registers_initialization` (`{}`): this dictionary allows the user to initialize any register to a concrete value by referencing it by its name in the architecture (e.g., rax, rbx). ByteMatcher interprets this option after `concrete_stack_addresses` so that it is possible to overwrite the default value of those registers.

Launcher

The launcher script was created to speed up the testing of ByteMatcher and provides a handler to interact with the project from the terminal and to provide C code instead of bytecode. It offers four actions:

1. `clear`: deletes the binary files from the files folder. It is possible to target single files or delete them all at once.
2. `compile`: compiles all the C files from the files folder. It is possible to target single files or compile them all at once.
3. `run`: initializes an instance of `Executor` with the provided C file.
4. `compare`: runs two instances of `Executor` with the provided C files and compares them with the given `EquivalenceLevel`.

All the actions allow the user to choose the architecture for the bytecode. The `run` and `compare` actions automatically compile the C file only if the respective binary file does not exist or if the last update of the source is more recent than the one of the binary.

5 | Experimental Validation

This Chapter presents the steps performed to validate the implementation of ByteMatcher along with all the details necessary to reproduce the results. The first half illustrates the general goals of the validation procedure, offering information about the setup in which the tests run. The second half shows the detail of each trial, from the composition of the test dataset to the results of the executions.

5.1. Goals of the Validation

The trials outlined in the following Sections want to prove the validity of the current implementation of ByteMatcher concerning two particular claims:

1. ByteMatcher can correctly recognize the semantic equivalence of two bytecodes using dynamic analysis techniques.
2. ByteMatcher can detect and report (possible) buffer overflow vulnerabilities when symbolically executing a bytecode.

Testing these claims requires designing two trials, each employing test cases from different source datasets tailored to the trial's requirements.

Besides testing whether the implementation of ByteMatcher is valid, the analysis of the trials' results aims to set a direction for a future extension of this project by discovering both the conditions in which ByteMatcher performs better and the aspects that need to be enhanced.

5.2. Experimental Setup

This Section describes the specifications of the system used to perform the trials. It also presents the approach to split each test dataset into smaller batches whose results can be combined to ensure a smooth execution of all the test cases.

5.2.1. Environment

The machine performing the experiments presents the hardware specifications outlined in Table 5.1. The operating system running on the machine is a 64-bit version of Ubuntu (5.15.0-58-generic).

Specification	Value
Processor	2×2GHz quad-core CPUs
Architecture	x86_64, little-endian
Memory	16 GB
Disk	32 GB (15 GB for / partition)

Table 5.1: Hardware specification for experimental validation.

In order to execute ByteMatcher, and therefore the trials, the system has Python installed. The tests run in an isolated Python environment that contains all the libraries required by ByteMatcher and specified in the project's repository.

5.2.2. Testing Approach

Both trials use the same testing platform implemented in Python to extract the test cases from the test datasets, execute them using ByteMatcher, and report the outputs. The platform generates two output files named `results.csv` and `errors.csv` in which it stores respectively the result of correctly terminating test cases and the execution traces of the ones that reach an error state.

Each test case is independent of the rest of the dataset by design so that the platform can work on fixed-size batches of random test cases instead of the whole dataset. This strategy allows the platform to interleave the execution of test cases from both trials. Moreover, the platform immediately records the output of a test (result or error) on file so that the execution can restart without losing the tests completed up to that point if a failure occurs.

The testing platform exploits all the machine cores to perform up to 8 test cases in parallel thanks to the `multiprocessing` library that instantiates a managed pool of workers. Each worker executes tasks on a different processor. Whenever a worker is free, the pool loads a test case from the current batch and assigns it to the worker. Each worker can consume up to **180 seconds** of CPU time and **4 GB** of memory. These limits, selected empirically, are

the ones that minimize the number of errors while keeping a reasonable execution time for each batch. If the worker exceeds any threshold, its task is aborted with a `TimeoutError` or a `MemoryError`, respectively.

5.3. Semantic Equivalence

5.3.1. Goal and Strategy

This first trial aims to collect enough data to prove that the approach implemented by ByteMatcher can classify whether two bytecodes are semantically equivalent with enough confidence. The execution of each test case incorporates two steps:

1. Execute two bytecodes from the test dataset with the default configuration.
2. Compare the results of the two executions using the default equivalence level (`Lite`).

The result of each test also provides insights into the symbolic execution of the two bytecodes, like the number of symbolic loops summarized or the number of function calls.

This trial comprises two experiments using a different test dataset to cover the full spectrum of possible results.

The first experiment compares pairs of bytecodes compiled from the same source code using different optimization flags. Under these settings, it is impossible to have false positive or true negative results since the functions under comparison are always supposed to be equivalent.

For this reason, the second experiment compares pairs of random functions that should not be equivalent (more on this in the following Subsection). Since this experiment cannot present true positives or false negatives, it ensures that combining the two experiments will fully populate the spectrum of possible results.

5.3.2. Dataset

The construction of the source dataset for this trial starts from the list of C++ projects also used to validate [3]. The list contains the URLs of 556 GitHub repositories. Considering only those projects that allow changing the optimization flag during compilation reduces the number of repositories analyzed to 431. Each "controllable" project is compiled for the x86_64 architecture using four optimization flags: `-O0`, `-O2`, `-O3`, and `-Os`. A script employs Angr [21] to extract the bytecode of each function, along with its name and memory address, from every binary. The script uses the name of each function to

match its bytecode across optimization levels.

The source dataset of this trial is a `csv` file that contains only functions presenting all four optimization variants and whose bytecode is longer than 6 bytes (the length of prologue and epilogue combined for x86_64 functions). This dataset contains 61 124 rows structured as in Table 5.2. Every row uses a concatenation of repository, binary, and function information as a global identifier and stores each variant in four consecutive columns.

Column	Description
Global ID	String of type "Author_Repo_Binary_Function".
Flag	Optimization flag of variant 1.
Address	Starting address of variant 1.
Size	Size in bytes of variant 1.
Bytecode	Binary representation of variant 1.
...	Information of variant 2.
...	Information of variant 3.
...	Information of variant 4.

Table 5.2: Row structure of the source dataset for the first trial.

Experiment 1

For the first experiment, test cases only compare pairs of bytecodes from the same function (i.e., from the same row). For every row of the source dataset, six possible combinations of two bytecodes exist. The six rows extracted from the same function share the same test group identifier that, combined with the flag of the two variants, provides a unique identification of the test. This first test dataset contains 366 744 test cases and is structured as in Table 5.3.

Column	Description
Test group ID	Global identifier of the function.
Flag	Optimization flag of function 1.
Address	Starting address of function 1.
Size	Size in bytes of function 1.
Bytecode	Binary representation of function 1.
...	Information of function 2.

Table 5.3: Row structure of the test dataset for experiment 1.

Experiment 2

The second experiment instead requires comparing pairs of bytecodes that are supposedly different. To ensure this characteristic, pairs of functions are selected randomly from the source dataset, independently of the optimization flag. In order to rule out the possibility of comparing equivalent bytecodes, the script excludes couples with the same name (checks only the function name, not the whole identifier) or with a perfectly matching bytecode. In this case, the number of possible test cases is enormous and is limited to 366 744 to match the size of the previous test dataset. Each test case follows the structure of Table 5.4. The test group identifier is the concatenation of the identifiers of the two functions under analysis. In this case, finding all possible variants in the test group is not guaranteed.

Column	Description
Test group ID	String of type "F1-GlobalID_F2-GlobalID".
Flag	Optimization flag of function 1.
Address	Starting address of function 1.
Size	Size in bytes of function 1.
Bytecode	Binary representation of function 1.
...	Information of function 2.

Table 5.4: Row structure of the test dataset for experiment 2.

5.3.3. Results

This Subsection presents the results of the first trial to validate this research. The goal is to demonstrate that ByteMatcher, as currently implemented, can effectively predict

whether two bytecodes are semantically equivalent. This trial includes two experiments for statistical reasons:

- The first experiment compares bytecodes compiled from the same source code with different optimization flags. The ground truth is always "Equivalent". Hence the prediction is either a true positive or a false negative.
- The second experiment compares bytecodes extracted pseudo-randomly from the source dataset. The ground truth is always "Not equivalent". Hence the prediction is either a true negative or a false positive.

The total number of test cases executed for the first experiment is 129 726. The tests compare the code of 21 621 functions compiled with four optimization flags (i.e., 6 test cases for every function). Of those, 97 169 (74.90%) result in a prediction from ByteMatcher. The first half of Table 5.5 summarizes the errors that prevented the rest of the test cases from terminating.

The second experiment, instead, produced 80 411 predictions from the 123 762 test cases executed (64.97%). In this case, the test cases compare the code of functions extracted randomly from the source dataset (with some checks to avoid checking undisputably equivalent functions). The second half of Table 5.5 shows the distribution of errors for this experiment.

	Experiment 1		**Experiment 2**	
Error type	**Occurrences**	**% of tests**	**Occurrences**	**% of tests**
`TimeoutError`	30 630	23.61%	41 091	33.20%
`MemoryError`	349	0.27%	105	0.08%
Other	1 578	1.22%	2 155	1.74%
Total	**32 557**	**25.10%**	**43 351**	**35.03%**

Table 5.5: Distribution of the errors reported during the first trial.

In both cases, `TimeoutError` dominates the table. It is, in fact, the cause of 94.08% and 94.79% of all the errors, respectively. The results (presented later) show that the loop summarization algorithm is the most likely cause of this massive contribution. In particular, some types of symbolic loops are too complex to be summarized from the current implementation of ByteMatcher. The threshold limiting the processor time (180 seconds) does not influence the outcome. In fact, during tests with up to 12 minutes of limit, the ratio of timeouts to the total number of tests was never lower than 20%.

`MemoryError`, on the other hand, only appears in 1.07% and 0.24% of the errors, respectively. This result is unsurprising given that the maximum memory for each test case is 4 GB and that the execution of ByteMatcher is not particularly memory-intensive.

The rest of the errors, classified in the table as "Others", consist mainly of failures related to `claripy`'s SMT solver and the `CFGFast` algorithm implemented by Angr [21]. This Subsection will not analyze them in detail, given their scarcity.

Besides the boolean value that represents whether the two bytecodes are semantically equivalent according to ByteMatcher, each test case provides insights into the symbolic execution of both bytecodes in the form of the number of symbolic loops and function calls detected. Table 5.6 shows the structure of each row of the `results.csv` file that contains the outputs.

Column	Description
Test group ID	Depends on the experiment.
Flag	Optimization flag of function 1.
Loops	Number of symbolic loops in function 1.
Calls	Number of symbolic loops in function 1.
...	Information of function 2.
Prediction	Whether the functions are semantically equivalent.

Table 5.6: Structure of each row of the result dataset for the first trial.

These results provide a new perspective on the outcome of the trial. If the percentage of test cases with at least one function call is realistic (26.35% and 46.59%), the rate of tests with at least one symbolic loop throws up a red flag. The fact that only 0.13% and 0.12% of the cases report the presence of loop summaries (less than expected for real-world functions) points with high probability to the loop summarization algorithm as the main weakness of ByteMatcher. For this reason, it is essential to remember that the considerations in the rest of the Subsection might suffer from survivorship bias.

The collected data contributes to creating a new set of features outlined in Table 5.7 to understand whether, and in which measure, the characteristics of the input influence the correctness of ByteMatcher's prediction.

Feature	**Description**
Flags	Concatenation of the compilation flags of the two bytecodes.
Has Loops	True if the two bytecodes combined have at least one symbolic loop.
Same Loops	True if the bytecodes have the same number of symbolic loops.
Diff Loops	True if the bytecodes have a different number of symbolic loops.
Has Calls	True if the two bytecodes combined have at least one function call.
Same Calls	True if the bytecodes have the same number of function calls.
Diff Calls	True if the bytecodes have a different number of function calls.

Table 5.7: Test features for the first trial.

Table 5.8 presents each feature's correlation with the trial's outcome (true if the prediction matches the ground truth, false otherwise). Unfortunately, the data shows no decisive influence for the second experiment. However, the outcome of the first experiment presents a strong (negative) correlation with the Has Calls feature. As expected, the data confirms the assumption that the uninterpreted theory employed by ByteMatcher to handle function calls introduces errors when comparing functions compiled with different optimization flags.

Feature	**Experiment 1**	**Experiment 2**
Flags	0.10	-0.04
Has Loops	-0.05	0.01
Same Loops	0.05	-0.01
Diff Loops	-0.05	0.01
Has Calls	-0.81	0.26
Same Calls	0.55	-0.25
Diff Calls	-0.55	0.25

Table 5.8: Correlation of test features with the outcome of the first trial.

At this point, it is interesting to statistically analyze the results of the two experiments at the trial level. This analysis discards some of the results from the first experiment in favor of an aggregate result dataset balanced in terms of ground truth distribution. Therefore, the analysis considers 160 822 test cases, half from each experiment.

Table 5.9 shows the confusion matrix obtained by comparing ByteMatcher's prediction (P) with the ground truth (T). At a glance, most test cases lie on the principal diagonal containing the correct predictions (true positives and true negatives). This observation already provides reassurances about the performance of ByteMatcher.

In addition, a convenient characteristic is that each of the two columns represents the result of a different experiment. Comparing the content of the columns, it appears that ByteMatcher performs better in tasks that involve non-equivalent bytecodes. The most likely explanation for this behavior is that, as known, some of the techniques employed by ByteMatcher introduce approximations that make it easy to find discrepancies when comparing the execution of two bytecodes independently of the ground truth.

		T	
		Equivalent	**Not equivalent**
P	**Equivalent**	56 006 (34.82%)	5 892 (3.66%)
	Not equivalent	24 405 (15.18%)	74 519 (46.34%)

Table 5.9: Confusion matrix of the first trial.

Table 5.10 provides additional insight into the performances of ByteMatcher by presenting the most commonly used statistical indicators computed with the same inputs as the confusion matrix. These metrics confirm the observations made on the confusion matrix. The high precision score and the non-spectacular recall are compatible with the feeling that the ByteMatcher performs better in the test cases from the second experiment. Indeed, according to the results, ByteMatcher is more prone to misclassify equivalent bytecodes as non-equivalent rather than the opposite. The F1-score, which is lower than 80%, is a further endorsement of this trend.

On a positive note, the data reveals a reasonably good accuracy. This metric is the most straightforward performance measure. It is usually ignored by many during analysis since it is heavily susceptible to unbalances in the distribution of the (actual) classes in the result dataset. In this case, however, the accuracy is computed on a set of results explicitly constructed to present the same number of equivalent and non-equivalent bytecodes to provide a realistic measure of the performance of ByteMatcher. The value of this score gives reassurance on the techniques presented in this research and certifies that, in most situations, ByteMatcher provides the correct prediction.

Accuracy	Precision	Recall	F1
81.16%	90.48%	69.65%	78.71%

Table 5.10: Statistics of the first trial.

To summarize, this trial proves that ByteMatcher can detect the semantic equivalence of two bytecodes, employing dynamic analysis techniques and symbolic execution. The results of this trial are satisfying but leave room for improvement. In particular, the data gathered from the test cases show that the loop summarization algorithm is the main point of failure of the current implementation of ByteMatcher and should be the focus of further research.

5.4. Buffer Overflow Detection

5.4.1. Goal and Strategy

This trial wants to gather evidence to support the claim that ByteMatcher can recognize (possible) buffer overflows during the symbolic execution of a bytecode, in particular when applying a symbolic loop summary as explained in Subsection 3.2.4. Additionally, the trial wants to discover which classes of vulnerabilities trigger a response in ByteMatcher. Vulnerabilities are classified according to the Common Weaknesses Enumeration (CWE) standard [17].

This trial is composed of only one experiment. Each test case extracts one bytecode from the dataset and executes it with the configuration variable `stop_buffer_overflow` set to `true`. This setting instructs ByteMatcher to abort the execution with a `BuffOverflowException` whenever it reaches a state that might hint at a buffer overflow vulnerability.

The result of each test case is a boolean flag stating whether the symbolic execution ended with a `BuffOverflowException` or was completed normally (i.e., whether ByteMatcher classifies the bytecode as vulnerable).

5.4.2. Dataset

This trial analyzes binaries from the DARPA's Cyber Grand Challenge (CGC) [9] qualifying event. In particular, the programs come from a GitHub repository [13] that holds the source code of the challenges and provides, among other tools, a script to compile them in binary form automatically. The compilation process results in a pool of 116 challenge

binaries built for the x86_64 architecture using the `-O3` optimization flag (there is no particular reason behind the choice of this flag).

A script similar to the one presented in the first trial uses ANGR [21] to extract all the functions from the pool of binaries. Even in this case, the dataset contains only those functions whose bytecode is at least 6 bytes long. The script collects each function in a `csv` file that represents the test dataset of this trial. This dataset contains 25 032 rows, one per function, structured as in Table 5.11.

Column	Description
Test group ID	String of type “Challenge_Function”.
Flag	Optimization flag.
Address	Starting address.
Size	Size in bytes.
Bytecode	Binary representation.

Table 5.11: Row structure of the test dataset for the second trial.

Each CGC program includes a `README` file containing the list of vulnerabilities affecting the binary according to the CWE standard. This trial focuses on the classes of vulnerabilities described in Table 5.12. The trial uses the output of the test cases to understand which combination of vulnerabilities is recognized by BYTEMATCHER.

Vulnerability	Description
`CWE-119`	Improper Restriction of Operations within the Bounds of a Memory Buffer.
`CWE-120`	Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’).
`CWE-121`	Stack-based Buffer Overflow.
`CWE-122`	Heap-based Buffer Overflow.
`CWE-787`	Out-of-bounds Write.
`CWE-788`	Access of Memory Location After End of Buffer.

Table 5.12: CWE vulnerabilities considered in the second trial.

5.4.3. Results

This Subsection elaborates on the statistical results of the single experiment of this trial, showing that ByteMatcher correctly reports (possible) buffer overflows during the symbolical execution of a bytecode.

ByteMatcher produced a prediction for 21 717 of the 25 032 test cases composing the test dataset (86.76%). Table 5.13 summarizes the errors reported during the experiment.

Like in the previous trial, the `TimeoutError` dominates the table, contributing to 85.64% of the total number of errors, while the occurrences of `MemoryError` are irrelevant. The considerations made for the first trial on the nature of the errors still hold.

Error type	Occurrences	% of tests
`TimeoutError`	2 839	11.34%
`MemoryError`	8	0.03%
Other	468	1.87%
Total	**3 315**	**13.24%**

Table 5.13: Distribution of the errors reported during the second trial.

The statistical analysis considers multiple classifiers representing different ground truths to cope with each binary having more than one vulnerability. In this way, the experiment can discover which vulnerabilities are predictable using ByteMatcher. In total, from 6 classes of vulnerabilities, the analysis obtained 63 classifiers, but Table 5.14 only reports the top-performing ones. As a clarifying example, the ground truth for the classifier `C1` says that a binary is vulnerable if it presents at least one vulnerability among the ones in the table (`CWE-119`, `CWE-120`, `CWE-121`, `CWE-787`, `CWE-788`).

ByteMatcher predicts 56 out of a total of 116 binaries as vulnerable. ByteMatcher's prediction for a binary is positive (vulnerable) if at least one of the functions of that binary triggers a `BuffOverflowException`. However, only some functions can contribute to the prediction. The analysis must exclude standard C library functions [14] because they introduce a bias in the statistical results by triggering the recognition without being inherently vulnerable, for example, `memset`, `memcpy`, `strcpy` and `strncpy` combined would account for 72.47% of all the positive results.

The goal of ByteMatcher in this experiment is to predict the presence of buffer overflows in a bytecode. Therefore, it is safe to assume that false positives are preferable

ID	CWE Vulnerabilities
C1	CWE-119, CWE-120, CWE-121, CWE-787, CWE-788
C2	CWE-119, CWE-120, CWE-121, CWE-122, CWE-787, CWE-788
C3	CWE-119, CWE-120, CWE-121, CWE-122, CWE-787
C4	CWE-119, CWE-120, CWE-121, CWE-787
C5	CWE-120, CWE-121, CWE-787, CWE-788
C6	CWE-119, CWE-120, CWE-787, CWE-788
C7	CWE-119, CWE-120, CWE-122, CWE-787, CWE-788
C8	CWE-120, CWE-121, CWE-122, CWE-787

Table 5.14: Top-performing classifiers of the second trial.

to false negatives. In cases like this, the best classifier would be the one with the highest recall. However, a classifier could fool this metric by always predicting true to get the maximum score. For this reason, the top performer is the classifier with the highest F1-score. This indicator rewards classifiers with both high recall and high precision.

From now on, the analysis continues focusing on the best classifier `C1`. Nevertheless, it shows the results in Table 5.15 and Table 5.16 for all of the top classifiers to give an idea of the differences in performance among them. Underlined values in each column represent the best result across all the classifiers.

It comes without much surprise that ByteMatcher does not excel in detecting buffer overflows that happen in the heap (`CWE-122`). This behavior is explainable as ByteMatcher does not simulate the execution of function calls and, therefore, is unaware of operations revolving around dynamically allocated buffers.

Other than that, the statistics support the claim that ByteMatcher can detect buffer overflows using symbolic execution. It can correctly predict the presence or absence of a vulnerability 2 out of 3 times (66.38% of accuracy), making it a discrete classifier.

False positives amount to 16.38%, which is less than expected considering that ByteMatcher detects vulnerabilities analyzing functions without any context. On the other hand, the best guess to explain the 17.24% of false negatives is that the misclassification happens on functions with vulnerabilities outside of symbolic loops or connected to a variable not recognized by the current implementation of the summarization algorithm.

ID	True positives	False positives	False negatives	True negatives
C1	37 (31.90%)	19 (16.38%)	20 (17.24%)	40 (34.48%)
C2	41 (35.34%)	15 (12.93%)	30 (25.86%)	30 (25.86%)
C3	40 (34.48%)	16 (13.79%)	30 (25.86%)	30 (25.86%)
C4	35 (30.17%)	21 (18.10%)	20 (17.24%)	40 (34.48%)
C5	34 (29.31%)	22 (18.97%)	19 (16.38%)	41 (35.34%)
C6	30 (25.86%)	26 (22.41%)	11 (9.48%)	49 (42.24%)
C7	34 (29.31%)	22 (18.97%)	22 (18.97%)	38 (32.76%)
C8	37 (31.90%)	19 (16.38%)	29 (25.00%)	31 (26.72%)

Table 5.15: Results of the top-performing classifiers of the second trial.

ID	Accuracy	Precision	Recall	F1
C1	66.38%	66.07%	64.91%	65.49%
C2	61.21%	73.21%	57.75%	64.57%
C3	60.34%	71.43%	57.14%	63.49%
C4	64.66%	62.50%	63.64%	63.06%
C5	64.66%	60.71%	64.15%	62.39%
C6	68.10%	53.57%	73.17%	61.86%
C7	62.07%	60.71%	60.71%	60.71%
C8	58.62%	66.07%	56.06%	60.66%

Table 5.16: Statistics of the top-performing classifiers of the second trial.

To summarize, the second trial demonstrates that ByteMatcher can recognize (possible) buffer overflows during the symbolic execution of a bytecode. The results are incredibly encouraging and open the way to developing new solutions to improve this functionality. In order to increase ByteMatcher's performance on the vulnerability detection task, future research should focus on improving the summarization algorithm and the approach to handle function calls.

6 | Limitations

ByteMatcher is a complex project that spans multiple open research fields absorbing ideas from cutting-edge tools and techniques. For this reason, the implementation could not overcome some limitations that leave few gaps in the domain of bytecodes that ByteMatcher can analyze.

This Chapter briefly presents the issues regarding:

1. How ByteMatcher implements the memory model.
2. How ByteMatcher summarizes symbolic loops.
3. How ByteMatcher handles function calls.

ByteMatcher is technically cross-architecture, meaning developers can easily extend ByteMatcher's code to analyze code compiled for any CPU included in the `archinfo` library. The tool inherits this characteristic from Angr [21] and `pyvex` [20]. Nevertheless, given how the symbolic state is modeled (in particular, the processor registers), the comparison operation is only possible between two bytecodes compiled for the same architecture. Solving this problem requires either redesigning the memory model used for symbolic execution or devising a brand-new algorithm capable of comparing two bytecodes employing different register sets and memory structures.

Summarizing a loop is a non-trivial operation that requires performing an in-depth analysis of the code. The current summarization algorithm limits the types of loops that ByteMatcher can properly handle. In particular, ByteMatcher:

- Cannot summarize nested loops or loops with complex paths in their body.
- Cannot correctly handle loops whose guard is not IV dependent.
- Cannot account for the effects of loops on variables that are not IV.

ByteMatcher inherits these issues from the original algorithm [12], but other researchers [24, 25] claim that a solution already exists, even if it was not possible to replicate their results in this work.

Finally, one of the requirements of ByteMatcher is to analyze and compare bytecodes that are fragments of complete programs stripped of all symbols. This prerequisite explains why ByteMatcher does not follow function calls during the symbolic execution. Instead, ByteMatcher implements an algorithm to skip the execution of the call based on the uninterpreted functions theory. This choice becomes a problem in contexts where ByteMatcher needs to compare a bytecode that performs a function call with one that presents the same function inlined among its instructions (e.g., due to compiler optimization). It is difficult to imagine an alternative approach to handle function calls and overcome this limitation.

7 | Future Works

This research outlines the implementation of ByteMatcher, a prototype that aims to cover multiple open research topics with a single tool for symbolic execution. As such, there are a few areas in which ByteMatcher can undoubtedly improve.

The approach to symbolic loop summarization represents the area where it is possible to have the most impact. The algorithm implemented in ByteMatcher is relatively constrained in the type of loops that can handle adequately. Future research should focus on finding a way to improve the existing technique, extending it to the summarization of loops with complex control structures and broadening the definition of IV, or developing an entirely new approach that can cover the majority of (if not all) the cases.

Since the development focused mainly on proving that the technique described in this work is viable, ByteMatcher can exceed time and memory constraints when performing intensive computations. In particular, an optimized memory model would improve the overall performance. A pagination mechanism for the memory as in [4, 8] would reduce the time required to retrieve the tuples when performing memory reads. In addition, applying the state merge operation also during the symbolic execution whenever possible (in addition to collapsing the state tree at the end) would decrease the number of paths to analyze and the memory absorbed by ByteMatcher. Finally, implementing the solution already described in [5] would improve the performance of the memory model in the presence of memory-intensive operations.

Lastly, there is no technical reason why the component that performs the analysis of the bytecode (`Executor`) should also represent the result of the execution at comparison time. Detaching the two features requires considerable structural changes but would improve the clarity of the code, other than reducing the space required to store the results of each computation. To implement this idea, the symbolic execution of a bytecode should return an object that contains only the subset of the data required to carry out the comparison.

8 | Conclusions

This thesis presented the research and the efforts behind the implementation of ByteMatcher, a tool for symbolic execution that mixes different existing techniques and new approaches to determine the semantic equivalence of bytecodes. This work began with an extensive analysis of the state of the art to understand the main challenges that symbolic executors face. The research converged on four categories of problems: the memory model, the interaction with the environment, the explosion of the state space, and the resolution of sets of complex symbolic constraints.

The development of ByteMatcher tried to distill the best solutions from past research and to mix them with new approaches to obtain a symbolic executor that could dynamically analyze bytecodes and recognize semantic equivalence. Among those solutions stands a fully-symbolic memory model which tracks both processor registers and memory addresses. This model implements algorithms to handle symbolic operations without the need for concretization and without branching the execution path.

The focal point of ByteMatcher's implementation was an original approach to compare fully-symbolic memory models resulting from the execution of different bytecodes. This technique is the one used to decide on semantic equivalence.

ByteMatcher also implements a technique to produce symbolic summaries of loops that it uses to emulate their execution. Loop summaries represent the best solution to avoid creating infinite execution paths without putting a limit on the number of loop iterations simulated. Other relevant techniques included a methodology to exploit the loop summarization to detect (possible) buffer overflows in the bytecode and an algorithm to produce fingerprints for function calls. ByteMatcher uses these fingerprints to avoid the burden of simulating external functions during symbolic execution.

Two sets of experiments (trial) provided data that supports the validity of the techniques implemented in ByteMatcher. A first evaluation tested whether ByteMatcher could correctly classify pairs of bytecodes as semantically equivalent (or non-equivalent), while a second analyzed its performance in detecting vulnerabilities. Both trials showed positive results and provided exciting insights into ByteMatcher's implementation.

Ultimately, the trials pointed out a few sides of ByteMatcher that need improvement, the most prominent of which is the summarization algorithm. Although, in theory, it should provide a great advantage to symbolic execution, this advantage did not appear in real-world scenarios. Nevertheless, ByteMatcher proved to be a fresh approach to binary analysis and deserves to be the subject of further research to refine the implemented techniques and introduce new functionalities.

Bibliography

[1] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1083–1094, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568293. URL `https://doi.org/10.1145/2568225.2568293`.

[2] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), may 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL `https://doi.org/10.1145/3182657`.

[3] L. Binosi. Recognition of inlined binary functions from template classes. Master's thesis, Politecnico di Milano, 2019/2020. URL `http://hdl.handle.net/10589/174104`.

[4] L. Borzacchiello, E. Coppa, D. Cono D'Elia, and C. Demetrescu. Memory models in symbolic execution: key ideas and new thoughts. *Software Testing, Verification and Reliability*, 29(8):e1722, 2019. doi: https://doi.org/10.1002/stvr.1722. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1722`. e1722 stvr.1722.

[5] L. Borzacchiello, E. Coppa, and C. Demetrescu. Handling memory-intensive operations in symbolic execution. In *15th Innovations in Software Engineering Conference*, ISEC 2022, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450396189. doi: 10.1145/3511430.3511453. URL `https://doi.org/10.1145/3511430.3511453`.

[6] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855741.1855756`.

[7] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary

code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012. doi: 10.1109/SP.2012.31. URL https://doi.org/10.1109/SP.2012.31.

[8] E. Coppa, D. C. D'Elia, and C. Demetrescu. Rethinking pointer reasoning in symbolic execution. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 613–618, 2017. doi: 10.1109/ASE.2017.8115671. URL https://doi.org/10.1109/ASE.2017.8115671.

[9] DARPA. DARPA Cyber Grand Challenge, 2014. URL https://www.darpa.mil/about-us/timeline/cyber-grand-challenge. (accessed: 01-04-2023).

[10] Y. David, N. Partush, and E. Yahav. Similarity of binaries through re-optimization. *SIGPLAN Not.*, 52(6):79–94, jun 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062387. URL https://doi.org/10.1145/3140587.3062387.

[11] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

[12] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 23–33, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305624. doi: 10.1145/2001420.2001424. URL https://doi.org/10.1145/2001420.2001424.

[13] GrammaTech. CGC Challenge Binaries, 2014. URL https://github.com/GrammaTech/cgc-cbs. (accessed: 01-04-2023).

[14] IBM. Standard C Library Functions, 2021. URL https://www.ibm.com/docs/en/i/7.3?topic=extensions-standard-c-library-functions-table-by-name. (accessed: 01-04-2023).

[15] T. Kapus and C. Cadar. A segmented memory model for symbolic execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 774–784, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338936. URL https://doi.org/10.1145/3338906.3338936.

[16] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni. Safe:

Self-attentive function embeddings for binary similarity. In R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329, Cham, 2019. Springer International Publishing. ISBN 978-3-030-22038-9. doi: 10.1007/978-3-030-22038-9_15. URL `https://doi.org/10.1007/978-3-030-22038-9_15`.

[17] MITRE. Common Weaknesses Enumeration, 1999. URL `https://cwe.mitre.org`. (accessed: 01-04-2023).

[18] D. Molnar, P. Godefroid, and M. Levin. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium, NDSS*, pages 416–426, 2008.

[19] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250746. URL `https://doi.org/10.1145/1273442.1250746`.

[20] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, pages 1–1, 2015.

[21] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016. doi: 10.1109/SP.2016.17. URL `https://doi.org/10.1109/SP.2016.17`.

[22] D. Trabish and N. Rinetzky. Relocatable addressing model for symbolic execution. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 51–62, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380089. doi: 10.1145/3395363.3397363. URL `https://doi.org/10.1145/3395363.3397363`.

[23] M. Trtík and J. Strejček. Symbolic memory with pointers. In F. Cassez and J.-F. Raskin, editors, *Automated Technology for Verification and Analysis*, pages 380–395, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11936-6. doi: 10.1007/978-3-319-11936-6_27. URL `https://doi.org/10.1007/978-3-319-11936-6_27`.

[24] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 61–72, New York, NY, USA, 2016. Association for Com-

puting Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950340. URL https://doi.org/10.1145/2950290.2950340.

[25] X. Xie, B. Chen, L. Zou, Y. Liu, W. Le, and X. Li. Automatic loop summarization via path dependency analysis. *IEEE Transactions on Software Engineering*, 45(6): 537–557, 2019. doi: 10.1109/TSE.2017.2788018. URL https://doi.org/10.1109/TSE.2017.2788018.

List of Algorithms

3.1 Register write operation. 21
3.2 Register read operation. 21
3.3 Register merge operation. 22
3.4 Multi-byte memory write operation. 23
3.5 Single-byte memory write operation. 23
3.6 Multi-byte memory read operation. 24
3.7 Single-byte memory read operation. 24
3.8 Memory merge operation. 25
4.1 Handler for conditional branches in `ExecutorState`. 36
4.2 Handler for a loop header hit. 38
4.3 Routine to update the `IVTable`. 39

List of Figures

3.1 High-level overview of ByteMatcher's main components. 18
3.2 Representation of a program as a black box. 26
3.3 Definition of the machine's state in the black box representation. 26
3.4 Semantic equivalence in the black box representation. 26

4.1 UML class diagram of ByteMatcher. 30
4.2 Detail of the UML class diagram for `LoopSummary`. 32
4.3 Black box representation in ByteMatcher. 37

List of Tables

2.1 Definition of each variable in the IVT tuples. 13
2.2 Definition of each variable defining a loop guard. 14

3.1 Definition of each variable in the register tuples. 20
3.2 Definition of each variable in the memory tuples. 22

4.1 Annotations of uninitialized `BV` objects. 40

5.1 Hardware specification for experimental validation. 44
5.2 Row structure of the source dataset for the first trial. 46
5.3 Row structure of the test dataset for experiment 1. 47
5.4 Row structure of the test dataset for experiment 2. 47
5.5 Distribution of the errors reported during the first trial. 48
5.6 Structure of each row of the result dataset for the first trial. 49
5.7 Test features for the first trial. 50
5.8 Correlation of test features with the outcome of the first trial. 50
5.9 Confusion matrix of the first trial. 51
5.10 Statistics of the first trial. 52
5.11 Row structure of the test dataset for the second trial. 53
5.12 CWE vulnerabilities considered in the second trial. 53
5.13 Distribution of the errors reported during the second trial. 54
5.14 Top-performing classifiers of the second trial. 55
5.15 Results of the top-performing classifiers of the second trial. 56
5.16 Statistics of the top-performing classifiers of the second trial. 56

Acknowledgements

Il mio primo ringraziamento va ai miei genitori, Fabio e Cristina. Grazie infinite per il vostro supporto in questo percorso e per avermi insegnato che l'impegno e la dedizione ripagano sempre. Questo traguardo è anche vostro.

Grazie a tutti i miei amici, a quelli che ci sono sempre stati e a quelli che si sono aggiunti lungo la strada. Grazie per essermi stati vicini in questi anni e per aver reso quest'esperienza indimenticabile.

Un ringraziamento speciale va alla mia ragazza per essere stata al mio fianco nei momenti belli ma soprattutto in quelli brutti. Sei riuscita a risvegliare in me quell'ambizione che da troppo tempo era dimenticata e te ne sarò per sempre grato.

Infine, voglio concludere con un augurio al me stesso di domani. Ti auguro di avere sempre qualcosa che ti faccia appassionare e di non perdere mai quella curiosità e quella voglia di imparare che ti hanno portato fino a qui oggi.

Been a hell of a ride but
I'm thinking it's time to go