POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

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

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Lorenzo Fratus

Advisor: Prof. Mario Polino

Co-advisors: Armando Bellante, Lorenzo Binosi

Academic year: 2021-2022

---

1. Introduction

Computer science is at the foundation of modern society, in which technology plays a crucial role in every aspect of people's life. 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.

Researchers made several attempts to devise solutions to support developers, resulting in automated tools for test generation and binary code analysis. Regardless of the excellent performance in the respective tasks, these tools have more than one limitation, leaving some challenges like the detection of equivalent pieces of code unsolved.

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 work takes inspiration from the implementation of Angr [4] and improves it by introducing, among other things, a fully-symbolic memory model. This model represents the machine's state 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.

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 experimental evaluation results 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 valid-

ity 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.

2. Approach

ByteMatcher offers two main functionalities to developers and security experts one to test the semantic equivalence of bytecodes, the other to detect (possible) buffer overflows. Both need to analyze code dynamically through symbolic execution. ByteMatcher employs Angr [4], 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 Angr's behavior rather than implementing it from scratch.

2.1. 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 bytecode is lifted into an Intermediate Language (IL) that provides an abstraction over architecture-specific details, allowing the executor to symbolically execute code originally compiled for any processor in the same way.
Figure 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.

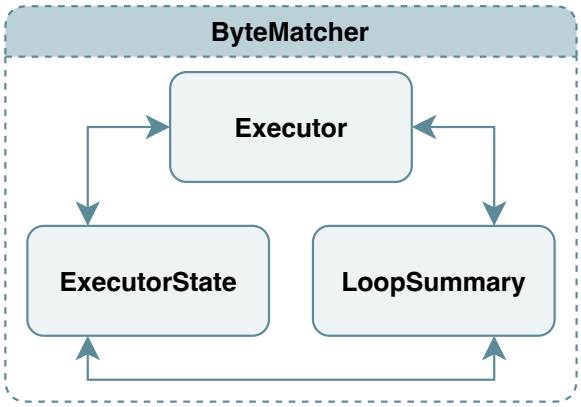

Figure 1: High-level overview of ByteMatcher's main components.

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 Angr's 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. Two notable cases are symbolic loops, managed with a unique algorithm implemented by `LoopSummary`, and function calls that ByteMatcher handles according to the uninterpreted function theory by building a fingerprint used as the return value of the function.

LoopSummary This component handles a particular case of conditional branches. Input-dependent loops represent one of the leading open challenges of symbolic execution. For this reason, this component implements a variation of the approach presented in [2] that uses symbolic execution to estimate the number of iterations and produce a reusable

logical summary of the loop. The summary is cached and used by `Executor` whenever it encounters a loop to update the `ExecutorState`.

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.

2.2. 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 fed into a Satisfiability Modulo Theories (SMT) solver to check whether they are compatible. Two 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.

2.3. 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 since their target might be user-controlled. This is especially problematic in the context of symbolic loops where also the number of iterations might be controllable. Since ByteMatcher analyzes out-of-context bytecodes and not entire binaries, detecting an overflow does not guarantee that the bytecode is vulnerable in real scenarios.

3. Experimental Validation

The trials outlined in the following 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. 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. 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 experiments of each trial run on a machine with 8 cores (2×2GHz quad-core CPUs), 16 GB of RAM, and 32 GB of disk. The operative system of the machine is a 64-bit ver-

sion of Ubuntu (5.15.0-58-generic). 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 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.

3.1. Semantic Equivalence

The execution of each test case to validate the first claim 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`).

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

Dataset The source dataset for this trial contains bytecodes extracted from a list of 431 C++ projects (inherited from [1]) compiled for the x86_64 architecture using four optimization flags: `-O0`, `-O2`, `-O3`, and `-Os`. The source dataset includes 61 124 bytecodes, with four variants each.
For the first experiment, test cases only compare pairs of bytecodes from the same function. This first test dataset contains 366 744 test cases.
The second experiment instead requires comparing pairs of bytecodes that are supposedly different. In order to rule out the possibility of comparing equivalent bytecodes, the script excludes couples with the same name or with a perfectly matching bytecode. In this case, the number of test cases is limited to 366 744 to match the size of the previous test dataset.

Results The total number of test cases executed for the first experiment is 129 726. Of those, 97 169 (74.90%) result in a prediction from ByteMatcher. `TimeoutError` accounts for 94.08% of the errors, while `MemoryError` is responsible only for 1.07% of them.
The second experiment produced 80 411 predictions from the 123 762 test cases executed (64.97%). `TimeoutError` accounts for 94.79% of the errors, while `MemoryError` is responsible only for 0.24% of them.
The threshold limiting the processor time (180 seconds) was empirically proven not to influence the outcome. The data points to the loop summarization algorithm as the leading cause of `TimeoutError` since only 0.12% and 0.13% of the results of the experiments report the execution of at least one symbolic loop.
Among the features that ByteMatcher reported along with its prediction, only one presents a clear correlation with the output of the first experiment. In particular, the Has Calls feature (true if the test reports at least one function call) has a correlation score of -0.81, highlighting that the uninterpreted function theory might introduce errors across versions of the same bytecode.
Table 1 shows the confusion matrix obtained by comparing ByteMatcher's prediction (P) with the ground truth (T). 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. Comparing the content of the columns, it appears that ByteMatcher performs better in tasks that involve non-equivalent bytecodes (experiment 2). 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	
		EQ	**NE**
P	**EQ**	56 006	5 892
	NE	24 405	74 519

Table 1: Confusion matrix of the first trial.

Table 2 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. The high precision score and the non-spectacular recall are compatible with the feel-

ing that the ByteMatcher performs better in the test cases from the second experiment. Indeed, according to the results, it 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 of 81.16%. This metric is the most straightforward performance measure that, when computed on a set of results explicitly constructed to present the same number of equivalent and non-equivalent bytecodes, provides 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.

Accur.	Prec.	Recall	F1
81.16%	90.48%	69.65%	78.71%

Table 2: 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 show that the loop summarization algorithm should be the focus of further research.

3.2. Buffer Overflow Detection

In addition to validating the second claim, this trial wants to discover which classes of vulnerabilities, classified according to the CWE standard, trigger a response in ByteMatcher. This trial is composed of only one experiment. Each test case 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`.

Dataset This trial analyzes binaries from the DARPA's CGC qualifying event. In particular, the programs come from a GitHub repository [3] 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. This dataset contains 25 032 different bytecodes. Each CGC program includes a `README` file containing the list of vulnerabilities affecting the binary. This trial focuses on the following classes of vulnerabilities: `CWE-119`, `CWE-120`, `CWE-121`, `CWE-122`, `CWE-787`, and `CWE-788`. The trial uses the output of the test cases to understand which combination is recognized by ByteMatcher.

Results ByteMatcher produced a prediction for 21 717 of the 25 032 test cases composing the test dataset (86.76%). Like in the previous trial, `TimeoutError` contributes to the majority of the total number of errors (85.64%), while the occurrences of `MemoryError` are irrelevant (0.03%). The considerations made for the first trial on the nature of the errors still hold.
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 because they introduce a bias in the statistical results by triggering the recognition without being inherently vulnerable.
The statistical analysis considered multiple classifiers representing different ground truths to cope with each binary having more than one vulnerability. In total, from 6 classes of vulnerabilities, the analysis obtained 63 classifiers. Table 3 presents the confusion matrix for the top-performing classifier `C1`. The ground truth for the classifier `C1` says that a binary is vulnerable if it presents at least one vulnerability among: `CWE-119`, `CWE-120`, `CWE-121`, `CWE-787`, and `CWE-788`.

		T	
		EQ	**NE**
P	**EQ**	37	19
	NE	20	40

Table 3: Confusion matrix of the second trial.

False positives amount to 16.38%, which is less than expected considering that ByteMatcher analyzes functions without 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. Other than that, the statistics outlined in Table 4 support the original claim. ByteMatcher can correctly predict the presence or absence of a vulnerability 2 out of 3 times (66.38% of accuracy), making it a discrete classifier.

Accur.	Prec.	Recall	F1
66.38%	66.07%	64.91%	65.49%

Table 4: Statistics of the second trial.

To summarize, this 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.

4. 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. A preliminary analysis of the state of the art revealed four main challenges that symbolic executors face: 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 registers and memory addresses, handling 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 and decide on semantic equivalence. ByteMatcher also implements a technique to produce symbolic summaries of loops. Loop summaries represent the best solution to avoid infinite execution paths without a reduction of the completeness of the analysis. 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.
Two sets of experiments (trials) provided data to support the validity of the techniques implemented in ByteMatcher. 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. 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.

References

[1] Lorenzo Binosi. Recognition of inlined binary functions from template classes. Master's thesis, Politecnico di Milano, 2019/2020.

[2] Patrice Godefroid and Daniel 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.

[3] GrammaTech. CGC Challenge Binaries, 2014. (accessed: 01-04-2023).

[4] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.