Automatic Vulnerability Detection: Using Compressed Execution Traces to Guide Symbolic Execution

Nuno Sabino
nuno.sabino@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2019

Abstract

Automatic Vulnerability Detection (AVD) is an area of research responsible to find how to automatically detect vulnerabilities in a certain program. There are many techniques to analyze a program. One of them, called symbolic execution, allows to execute all possible program paths using symbolic variables instead of concrete values. In this thesis, we develop on a version of AVD where we have a set of inputs that trigger vulnerabilities (bad inputs) and a set of inputs that demonstrate functionality (good inputs). This thesis contributions are three-fold. First, we survey the current state-of-the-art techniques on software analysis, focused on the security aspect. Second, we show how to implement the mentioned techniques in a language-independent way and what were the problems we faced while building such a tool for program analysis and the reason why we chose one solution over the others. Finally, we show how to run a prioritized version of symbolic execution, given tests that exercise some part of the program, that is able to traverse program paths similar to those run by the test with the assistance of a dynamic similarity coefficient. We have evaluated our tool by analyzing its efficiency and efficacy in finding vulnerabilities in a dataset of DARPA challenges used in the Cyber Grand Challenge competition. We successfully analyzed 186 binaries in that dataset, where we accurately found 118 vulnerabilities out of a total of 590 intended ones. Additionally, we compared the application of our dynamic similarity coefficient with Ochiai’s and another one used in an automatic patch generation tool called Staged Program Repair. We show that our dynamic similarity coefficient significantly outperforms the others, enabling us to analyze a greater number of binaries and therefore finding more vulnerabilities. Our dynamic heuristics demand that we keep the execution traces in memory while running symbolic execution, so we have developed a specialized algorithm to compress traces using redundancy from loops and recursive calls, which achieved a compression factor of 36.6, compared with the original trace size.

Keywords: Automatic Vulnerability Detection, Security Tests, Software Analysis Techniques, Symbolic Execution

1. Introduction

1.1. Motivation

Devices like mobile phones, laptops or IOT devices are becoming increasingly more complex. That complexity makes the process of finding bugs in a software incredibly hard, therefore allowing security vulnerabilities to exist. Companies strive for keeping up with hackers in this race to find vulnerabilities which impact is growing due do the sprouting connectivity of all devices nowadays. That crescent connectivity augments the attack surface in a dangerous way and companies are increasingly worried about the prospect of being attacked. The WannaCry virus, a malware that affected more than 400,000 machines caused a total of around $4 billions in damage [8]. According to a study made by Accenture, the average cost of malware attack for a company is $2.4 million. Another report made by a software testing company called Tricentis revealed that software failures caused a total of $1.7 trillion in financial losses in 2017 [3]. What can we do about this? How can we timely protect companies and their products from sustained attacks that cause such a financial impact? This thesis is born of an attempt to automatize the process of finding security bugs in software in order to reduce as much as possible the time it takes to patch a security vulnerability.

1.2. Topic Overview

In software testing, when a failure of a program is observed one must first find the bug that originated that failure. It all comes down to finding the specific part of the code that should be present or the
specific part that should be removed or modified, in order to achieve the required functionality. The process of fixing security bugs is not that different from the process of fixing functional bugs. A root cause must also be found and in this case, it is the part of the code that does not respect the security standards defined by the developers, the community and the context of the project at hand.

Finding the root cause of a failure is a very cumbersome job and frequently takes a lot of time (and budget) to find all necessary bugs to achieve a reasonably functional piece of software. Syntax errors can be easily fixed by going to the line of code that failed and modify it so that it is according to the rules of the language, which are usually well defined. The problem appears when we need to fix semantic errors. In that case, the compiler does not know what is wrong and what is right on that level and is unable to give details about the root cause of the bug. The compiler does not know what the developer intended to program, it can only see language rules. That means we can only see a program is failing by observing its results. Matching an incorrect behaviour with the respective incorrect piece of code takes a lot of time and energy. On this thesis for instance we spent a month improving code until it didn’t crash all the time and sometimes the problem wasn’t even in our code, but on one of our dependencies. It is obvious for a software tester that every software has bugs, and we need something that help us with the process of finding the root cause of a bug and automatically fix it in a timely and accurately manner.

There is some important work on automating vulnerability discovery. The most frequently used techniques are fuzzing, dynamic taint analysis, symbolic execution or a mixture of them. Fuzzing is a promising technique that analyzes the program by feeding it somewhat random inputs that hopefully will crash the program or cause other unintended behaviour. This technique is able to find crashing inputs very fast but has a poor path coverage, though it is possible to use genetic techniques to improve the overall coverage by conserving inputs that have properties corresponding to paths that we want to explore further. Dynamic taint analysis is able to mark variables that satisfy some special user defined property. Those marked variables are called tainted variables. Operations with tainted variables can propagate taint to the results. This technique can be used for checking whether the user can control something that he wasn’t supposed to control by tainting input variables and checking for taint on the respective critical variables. For instance, we can check for format string vulnerabilities by observing if the format string argument (i.e. either the address or the contents pointed by the address) is tainted. Of course, this technique alone can not be used to find a reasonable variety of vulnerabilities. Sometimes, in order to compromise a system we must reach a very specific vulnerability that is only triggered in a state that satisfies very specific restrictions. It is easy to imagine a program that is only vulnerable if some variable x has value 1. If we want to explore every possible state that the program can have, symbolic execution is the technique we should use. It replaces all unknown values (e.g. user input, random variables, etc) with symbolic variables and performs all operations using those variables. An operation between symbolic variables yields another symbolic variable. We can use this to solve many problems but unfortunately, this technique has the inherent issue that the number of possible states is gigantic, thus exploring all of them is usually unfeasible. In this thesis, we are going to explore a few techniques to improve symbolic execution and tackle that state explosion problem.

The event that was perhaps the eye-opener to the capability of automatic systems to generate patches and exploits was the Cyber Grand Challenge (CGC) where a number of teams developed their own systems that automatically exploited and patched programs and they made those systems compete against each other. CGC was organized by the Defense Advanced Research Projects Agency (DARPA) to push forward the state-of-the-art in automatic patch generation. Even though most of the generated patches were generic (e.g. applying protections similar to canaries after critical data on binaries), the systems revealed to be highly capable at exploiting. The first, second and third place winners of CGC were rewarded with $2M, $1M and $750,000 respectively, which reflects the relevance that is given to research in this area. Due do that competition, many tools similar to ours were developed and perfected, and the vulnerable binaries that appeared in the competition are now available to anyone who wants to evaluate the performance of such a tool on real-world like services.

1.3 Objectives

This thesis has three objectives: First, we study the state-of-the-art techniques on program analysis and the most commonly found vulnerabilities existent on binaries, which are the ones we want to detect. Second, we design and implement a system that is able to automatically detect vulnerabilities and we describe the problems that were faced and the solutions we have found. Finally, we will describe our dynamic similarity coefficient which we used to prioritize symbolic execution by using execution traces from malicious inputs and functionality tests. We evaluated our tool by operating on a dataset from DARPA of 246 challenges, each containing one or more vulnerabilities, which were used in the CGC.
Our main contribution is the development of a new dynamic similarity coefficient which can be used to prioritize the symbolic execution of program paths in order to more efficiently perform concolic execution. Using this similarity coefficient, we implemented a scalable version of symbolic execution that was able to accurately detect 118 vulnerabilities, out of a total of 590 in a dataset used in the cyber grand challenge consisting of binary programs.

1.4. Thesis Outline
We will describe the background theory behind this work in Chapter 2, including state-of-the-art techniques and systems for analyzing programs. In Chapter 3, we present an overview of the design and architecture of our system and the justification for our implementation choices. In Chapter 4, we describe the experimental evaluation of our system and a summary of the DARPA CGC dataset. Finally, we reason about our findings and conclusions at Chapter 5.

2. Background
SPR [10] is an automatic patch generation tool published in 2015. They also work in a context where they have access to a set of positive (functional) tests and negative (not passing) tests. Intuitively, the repairs should be placed in the instruction of the code that is most likely to contain the bug and the techniques used to find that instruction are called fault localization techniques.

One can explore how to best relate the execution traces of negative and positive tests to find the instruction responsible for the bug as accurately as possible. The result would be a formula, commonly called similarity coefficient that uses properties about the executed tests to compute the likeliness for an instruction to be responsible for a defect.

Abreu et al. [5] describe and evaluate different similarity coefficients used in tools like Pinpoint [6], Tarantula [9] and AMPLE [7]. From the evaluated similarity coefficients, Ochiai’s, which is used for computing genetic similarity, got the best results. Each coefficient is a formula $S_j$ that computes how likely it is for instruction (or block of instructions) $j$ to contain the error and typically uses four counters:

- $a_{11}$ - How many tests with errors included the execution of block $j$
- $a_{10}$ - How many tests without errors included the execution of block $j$
- $a_{01}$ - How many tests with errors did not include the execution of block $j$
- $a_{00}$ - How many tests without errors did not include the execution of block $j$

Ochiai’s formula is given by $\frac{a_{11}}{\sqrt{(a_{11} + a_{01})(a_{11} + a_{10})}}$ and only uses three of the typical counters. SPR on the other hand only uses the $a_{11}$ and $a_{00}$ counters from the typical ones, and also one extra counter $b = \sum_{t \in T} r(j, t)$ where $r(j, t)$ is the last index of the execution trace acquired from test $t$ where instruction $j$ appears. Their similarity coefficient doesn’t output the likeliness of instruction $j$ to contain a bug but it is actually used to compare two instructions $j$ and $j'$ to see which one is the most likely to contain a bug. Assume that we have three extra counters - $a'_{11}, a'_{00}$ and $b'$ - that means exactly the same as the previous counters but for instruction $j'$. Their formula is given as follows:

$$\text{prior}(j, j') = \begin{cases} 
\text{True} & a'_{11} > a'_{11} \\
\text{True} & a'_{11} = a'_{11}, a'_{00} > a'_{00} \\
\text{True} & a'_{11} = a'_{11}, a'_{00} = a'_{00}, b > b' \\
\text{False} & \text{otherwise}
\end{cases}$$

Where prior$(j, j')$ yields True if $j$ is more likely to contain a bug than $j'$.

This kind of formulas can complement symbolic execution by making it execute only the paths followed by the malicious inputs (or non passing tests) and not followed by the passing tests. This will result in finding the root cause for the POVs at hand, instead of using pure symbolic execution to detect every possible bug. The trick is to use fuzzing to generate inputs that cause unintended behaviour and then use symbolic execution to find the root cause for the observed problems. We assume that fuzzing was already performed and in this thesis we will only worry about how to most efficiently use the execution traces to prioritize symbolic execution.

3. Solution
In this thesis, we will focus on analyzing x86 binaries and we assume we do not have access to the source code. The binaries will be the same ones used in the Cyber Grand Challenge (CGC) competition, a dispute between tools that automatically exploit and patch binaries. This dataset is a collection of 246 challenges where each challenge consists of a set of binaries, a set of functionality tests (to assert that the challenge behaves correctly) and a set of proof-of-vulnerability (POV), which are other binaries that interact with the respective service and prove that it is indeed vulnerable, either by crashing it in a controlled way or by leaking memory from a predefined page that is always allocated on every challenge.

In our analysis, we start by taking the target binary and lift it to an intermediate representation which will ease further analysis work. Notice that in order for our tool to work for some language (besides x86) one must only implement a lifter from that language to BIL, which is the intermediate representation that we will use. The remaining analysis
is done on top of BIL, therefore our design is independent of the chosen language as long as there is a lifter from that language to BIL.

We proceed to instantiate one memory - the initial state - that will start execution from the first instruction of the main function. We will have a stack of memories, only containing the initial state at the beginning, which will hold the states during symbolic execution. The stack will hold all states that have been discovered but haven't been explored yet. Symbolic execution is performed using a DFS search, meaning that every time we find a reachable state, we place it on the top of the stack and the next state to execute will also be taken from the top of the stack (LIFO). A state is reachable when there is at least one input that makes the program execution reach that state, for some assignment on the program variables.

We continuously pop states from the top of the stack until the stack is empty. For each state, we verify what is the next piece of code to execute. If it is a call to a function for which we already have implemented a summary that is able to simulate the execution of that function, then we call the summary instead. Otherwise, we interpret the code with our BIL Interpreter, which is able to reason about instructions of the intermediate language. In any case, further memories can be spawned due to symbolic execution, which will be inserted at the top of the stack, and their relative order depends on the priority of the respective blocks of instructions representing the next code to be executed on each of the states. The relative priority of two blocks of instructions depends on the value of applying our similarity coefficient to each block (higher value means higher priority). We only push a state to the stack if the restrictions on the respective symbolic variables are satisfiable. We use Z3 to reason about symbolic restrictions and check for state satisfiability.

We execute the program concretely with some inputs that test functionality and some exploits (POVs) which cause unintended behaviour on the program, and finally we acquire the respective execution trace which is a list of executed instructions by execution order. Those traces are then processed and used to compute the likeliness of some instruction to be responsible for the vulnerability in question. The states where the next instruction is the most likely to contain the vulnerability will have the most priority, thus our tool will attempt to execute them first. After executing the next instruction on the state, we call our safety policies which will check for security related inconsistencies in the current state.

3.1. BAP-IL Lifting
There are BAP lifter bindings for python, including an abstract data type (ADT) parser that uses the visitor pattern to simplify any analysis that implies writing specific behaviour for each instruction. To clarify, the BAP lifter yields the lifted instructions in an ADT format and we parse those instructions with the ADT parser by writing a visitor function for each instruction (e.g. Load, Store, Plus, Minus, among others). This allows to create our BIL Interpreter, which effectively executes BIL instructions and will be needed for the next steps.

We chose BAP-IL as our intermediate language for being explicit, self-contained, having bindings for python, it was used in the state-of-the-art vulnerability detection tool Mayhem, that won the CGC, and it has a well defined formal specification [1].

3.2. Summaries
We are interpreting binary code with python, meaning it will be much slower than normal. Some libc functions do a lot of work that we don't need to do. For instance, we don't need to execute exit (a function executed in a binary whenever it gracefully exits), we just need to terminate the program (or at least the current state). In addition to that, symbolically executing libc functions is generally not a good idea. Summaries are capable of operating with symbolic arguments in a smart, function specific way.

Summaries are also responsible for introducing symbolic variables. Summaries for fgets, gets, read, among other functions that receive input effectively introduce symbolic variables because we want to simulate every possible input that could be provided by the user.

3.3. Mixed Analysis
The second step in our analysis is taking the lifted code and actually run symbolic execution, either by executing BIL code via our BIL Interpreter or executing summaries. We load the instructions to execute, initialize and prepare a memory to execute the initial instructions in main and insert it in a memory stack. While there still is at least one memory in the memory stack, we pop the last memory (i.e. the one with higher priority) and we execute its next instruction. Both the summaries and the BIL Interpreter return a new (possibly empty) list of states which can actually be comprised of the original state itself, with an updated instruction pointer register. Finally, if we find a branch that spawns two memories in BIL Interpreter, a list with two elements will be returned and we add both states to the top of the stack according to their priority. We describe how to decide which memory should have the highest priority in section 3.6. Notice however that since we are executing a DFS, the recently created memories will go to the top of the stack anyway, its only their relative order that needs to be
decided. Note as well that it is easy to define a depth limit in order to execute an iterative DFS, by checking in the main loop if the current memory is too deep in the execution tree.

The symbolic execution per se is done in the inner workings of BIL Interpreter and in the summaries, since both the visitors in the BIL Interpreter and the summaries are prepared to receive symbolic operands and will act accordingly. Each time the BIL Interpreter finds a conditional instruction it first checks whether the condition can be true, false or both. If it can only be true or false it simply redirects the flow of execution to the body of the conditional statement. In the situation where it can be both true and false, which is possible in case the condition is symbolic, it creates two other memories and marks them as children of the original memory.

3.4. Safety Policies

Algorithm 1: Safety policy for arbitrary read

```
1: procedure DETECT
2: Require: code, the block of code that would be executed next
3: Require: interpreter, BIL Interpreter
4: Require: mem, the current memory (state)
5: if isLoad(code) then
6:     addr ← getLoadAddr(code)
7:     addr ← interpreter.computeExpression(addr)
8: if isSymbolic(addr) then
9:     return arbitraryReadDetected
10: User controls the address to load
```

We try to gather as much information as possible about the current state in order to make decisions about what to do next. It is not just a matter of collecting the restrictions on program variables, we also use dynamic taint analysis to keep track of variables that should not be leaked and we hold a few more structures that allow vulnerability detection. For instance, we have a table describing which addresses of the heap have been freed and which are meta-data, that helps us detect heap overflows or double free vulnerabilities more accurately.

A safety policy is a function (in this case, a python function) that is triggered on a certain BIL instruction and checks if the current program path is safe according to symbolic restrictions and how tainted are the variables. For instance, we can define a safety policy for the Load instruction as in Algorithm 1, which loads a value from memory accordingly to some address, stating that we should never load from a completely symbolic address (i.e. preventing arbitrary reads). A safety policy as we use in our system is very easy to implement as long as a vulnerability and a detection method can be formally defined. One simply has to use the easily accessible available information to decide whether a state is vulnerable or not.

3.5. Memory

We need to model the memory in a way such that it will be straightforward to support symbolic values for variables later. Each state will have its own memory and we started by implementing a memory as an individual python dictionary (which is implemented as a hash-table) where keys are addresses and values are bytes.

One solution that should reduce exponentially the amount of memory that we need to hold is to make each memory of a state depend on the memory of its parent state. The mentioned redundant information will only be kept in a single place (the state where the variables were defined) and each new memory only contains the differences in stored values relatively to its parent state.

3.6. Prioritization

A memory represents a state. Each memory has access to a mapping between addresses and bytes and another between registers names and 32-bit values (64-bit in case of x86_64). The value associated with the Instruction Pointer (IP) register points to the next instruction to be executed.

Our heuristics will be something that allow us to decide between one of two instructions which one has the highest priority. Assume we have access to a set of tests $H = \{ L_i \mid L_i \text{ is an execution trace} \}$. Each list $L_i$ corresponds to an ordered list of instructions that were executed by either a POV proving a vulnerability, or a functionality test that did not show any. Furthermore, we call $I^L_i$ to the address of the $i$th executed instruction in list $L$ and therefore when we say $I^L_i = I^L_j$ we are stating that both these instructions represent the same address (i.e. same value of IP) and thus the same instruction. On the other hand, if $L$ does not have at least $i$ instructions, then $I^L_i$ has a NULL value.

The goal here is to keep executing program paths as close as possible to what all the given exploits execute, on a single symbolic execution. Each memory has a state represented as the tuple $(H, S, currentInstr)$ where $H$ is our set of tests, shared between all states, $currentInstr$ represents the current instruction being executed and $S$ is a set of tuples representing the index of the current instruction on each test that is still being followed (at some point, we are probably going to diverge from all but one test). Formally, $S = \{ (L, i) \mid L \in H, I^L_i \in L, I^L_i = currentInstr \}$. If we have access to $n$ tests then $S$ will have at most that many elements. An instance $(L, i)$ will be removed from $S$ if $I^L_i = NULL$ which happens after executing the
last instruction from that test.

If we name the next instruction to be executed for a certain memory as \( \text{nextInstr} \), then \( \text{nextInstr} \) is on a direct path relative to test \( L \) if and only if \( \exists (L, i) \in S : I_{i+1}^L = \text{nextInstr} \). On the other hand, \( \text{nextInstr} \) is on a relaxed path relative to test \( L \) if and only if it is not on a direct path and \( \exists j, (L, i) \in S : I_{j}^L = \text{nextInstr} \) and \( i < j \). Upon execution of an instruction \( I \) the indexes on \( S \) are updated accordingly to reflect the first position after the previous index where the respective instruction is equal to \( I \) (if it is on a direct path it simply increments the previous index).

Imagine our program execution branches and we have two new memories that should be added to the stack. If memory \( M1 \) has an IP pointing to instruction \( I1 \) and memory \( M2 \) is pointing to \( I2 \), then the highest priority memory is the one for which the next instruction is in more direct paths relatively to the POVs and fewer direct paths relatively to the functionality tests. In case both instructions are on the direct path of the same number of tests, the tie is settled by the one with the most relaxed paths relatively to the bad tests.

Trivially, we prioritize the instructions that would be on a direct path relatively to the POVs, because those are the ones where the vulnerabilities are most likely to be. Similarly, being on a direct path relatively to a functionality test is something we want to avoid because there were no detected vulnerabilities during the execution of those. The novel part here to the best of our knowledge is the introduction of the relaxed path concept. The rationale for its existence is based on the observation that if we are following an execution trace generated by a POV and we are about to execute an instruction that even though it is not the next instruction as it was executed by the POV, it would execute it later, then that program path may be susceptible to a vulnerability too.

The computation of the most prioritized instruction to be executed in the symbolic execution is deeply related with the fault localization problem. We want, after all, to prioritize the instructions that are more likely to contain errors (in this case, vulnerabilities). The difference in our case is that the scoring is assigned dynamically instead of statically. The values of applying the similarity coefficient to each instruction is computed once in the beginning and then used throughout the analysis. We on the other hand, want the scores to be different depending on which instructions we have executed.

As an example, suppose we have two POVs and one good test \( H = \{POV1, POV2, GOOD\} \) and \( POV1 = [7, 8, 10], \) \( POV2 = [7, 9, 10] \) and \( GOOD = [7, 8, 11] \), therefore the good test executed the instructions with address 7, 8 and finally 11. In the beginning, we have \( S = \{(POV1, 1), (POV2, 1), (GOOD, 1)\} \) and \( \text{nextInstr} = 7 \). It is easy to see that \( L_{POV1}^1 = L_{POV2}^1 = L_{POV3}^1 = \text{nextInstr} = 7 \). If instruction 7 is a conditional jump, with a symbolic condition, where we can either jump to instructions 8 or 9, then we must create two new memories one for each instruction, and we need to see which of those two instructions should be prioritized. We can see that instruction 8 would follow a direct path on POV1 because \( L_{POV1}^2 = 8 \) and instruction 9 would follow a direct path on POV2 because \( L_{POV2}^2 = 9 \). Despite that, instruction 8 is also following a direct path on the good test, therefore we should prioritize instruction 9 because it is the one following fewer good tests.

3.7. Compressing execution traces
We can use PIN [11], a tool developed by Intel to dynamically instrument the binary in order to add code before each instruction. That code should store the value of the IP register (Instruction Pointer) in a new line in a trace file whenever that instruction is executed, effectively generating an execution trace.

One of the problems with execution traces is their size. We would like to keep the traces in memory during execution of the program but most programs execute tremendous amount of instructions for even simple inputs. Holding a list with Gigabytes of data in memory is not feasible. We need to take advantage of the repeating patterns inherent from the execution of loops and from recursive calls. With that in mind, we developed our algorithm for compressing execution traces and we now proceed to describe it.

From now on, we use the notation \( L[i : j] \) to represent all elements (or subgroups) that start at index \( i \) of list (or group) \( L \), all the way until the element with index \( j \), exclusively. Similarly, we use \( L[i] \) to represent the element of list \( L \) with index \( i \) and \( L[i :] \) represents all elements that start at index \( i \) until the end of the list or group. Consider \( L \) is the list of instructions in the trace that we are trying to compress. Adding two lists \( L1 + L2 \) yields another list containing all elements in \( L1 \) followed by all elements in \( L2 \).

We consider a suffix to be a tuple of three elements: The \( \text{start} \), the \( \text{end} \) and the \( \text{next} \). The \( \text{start} \) represents the index in \( L \) where the suffix begins. The suffix includes all instructions from \( \text{start} \) to \( \text{end} \). The \( \text{next} \) attribute represents the next instruction we would expect to see, if we were matching the suffix with the most recently processed sequence of instructions. \( \text{next} \) is always an index between \( \text{start} \) and \( \text{end} \).

Next, assume a \text{group} is a list of other groups
or ultimately instructions. A group has a property called size, which is the number of the subgroups that are included in that group, and multiplicity which means that the sequence of instructions represented by that group actually repeats that many times in the execution trace. We can compute the sequence of instructions represented by a group with the unfold operation. The unfold operation can be represented as follows:

\[
\text{unfold}(G) = \left\{ \begin{array}{ll} G.multiplicity \cdot \text{unfold}(G) & \text{if } G \in I \\ \sum_{j=0}^{\text{unfold}(G[j])} \text{unfold}(G[j]) & \text{otherwise} \end{array} \right.
\]

Unfolding a group is concatenating the unfolding of all its subgroups, and finally the repetition of that resulting sequence of instructions as many times as the multiplicity of the group. Take an example:

\[
\text{unfold}(\{ \text{multiplicity} = 2, \text{subgroups} = \{[I_2, \{\text{multiplicity} = 3, \text{subgroups} = \{I_1\}]\} \}) = \sum_{i=1}^{2} \sum_{j=1}^{3} \text{unfold}(I_2) + \sum_{i=1}^{2} \sum_{j=1}^{3} \text{unfold}(I_1)
\]

Note that adding two lists is not a commutative operation, because instructions in lists have order. The process we developed for compressing traces is described in Algorithm 2 and is quite simple.

In short, we keep a list of suffixes which are sequences of instructions which can be seen as the body of a loop (the part that repeats) for example. Each instruction is a suffix in itself (so as to detect repetitions of a single instruction) but suffixes will grow as the algorithm progresses. When all instructions in the suffix are matched, we can be sure that the sequence of instructions represented by that suffix effectively repeats and we merge both sequences into a single one.

The objective of this algorithm is to produce a group \( G \) such that \( \text{unfold}(G) = L \), therefore creating a structure that represents exactly the same as the original list but in a compact form. Of course, we want to be able to access information about the original list but in a compact form. We can compute the sequence of instructions represented by a group with the unfold operation. The unfold operation can be represented as follows:

\[
\text{unfold}(G) = \left\{ \begin{array}{ll} G.multiplicity \cdot \text{unfold}(G[j]) & \text{if } G \in I \\ \sum_{j=0}^{\text{unfold}(G[j])} \text{un} \text{fold}(G[j]) & \text{otherwise} \end{array} \right.
\]

Algorithm 2 Compressing traces

1: procedure MERGE
2: Require: \( G \), a group
3: Require: \((i, j)\), a tuple representing a subgroup (start, finish) of \( G \)
4: Require: \( L \), a non-empty list of instructions in the execution trace
5: procedure COMPUTE
6: result \leftarrow \text{EmptyGroup}
7: \( S \leftarrow \text{EmptyList} \) \quad \text{do} S is a list of suffixes
8: \( i \leftarrow 0 \)
9: \( sz \leftarrow \text{length}(L) \)
10: repeat
11: \( \text{result.addSubgroup}(I_i^j) \) \quad \text{for all } s \in S \quad \text{do}
12: \quad \text{if } s._\text{next} == I_i^j \text{ then}
13: \quad \quad s._\text{next} ++
14: \quad \quad if s._\text{next} == s._\text{end} \text{ then} \quad \text{If we have found the whole suffix}
15: \quad \quad \quad \text{merge(result, s.start, s.end)}
16: \quad \quad \quad S = S[0 : \text{indexOf}(s + 1)] \quad \text{Remove all smaller (thus invalid) suffixes}
17: \quad \quad \quad \text{Reset state and increase boundaries of all suffixes}
18: \quad \quad break
19: \quad else \quad \text{Reset state of suffix}
20: \quad \quad \text{if } I_i^j == s._\text{start} \text{ then}
21: \quad \quad \quad s._\text{next} = s._\text{start} + 1
22: \quad \quad \quad s._\text{end} = i
23: \quad \quad else
24: \quad \quad \quad s._\text{next} = s._\text{start}
25: \quad \quad \quad s._\text{end} = i + 1
26: \quad \quad append(\text{start} = i, \text{next} = i, \text{end} = i + 1) \quad \text{to } S \quad \text{for all instructions that are a suffix to } S
27: \quad end
28: \quad i \leftarrow i + 1
29: until i == sz
That would have to be done even if we hadn’t compressed the traces anyway, and performing this on the compressed structure is actually an optimization because we only have to check each instruction once (because repetitions are abstracted away in the multiplicity property).

It is easy to see that the algorithm described above is bounded by $O(n^2)$ with $n$ being the number of instructions in $L$. For each instruction, we iterate over all suffixes (which are, at most, as many as the number of instructions) and only one of the suffixes can cause a merge and a reset, both linear operations.

4. Results
The DARPA dataset we used consists of 246 challenges, each with one or multiple vulnerabilities. Each challenge comes with a set of proof-of-vulnerabilities that one can feed to the binaries to assert whether it is vulnerable. Each challenge also comes with the specification of a state machine that fully describes the behaviour of the challenge and can be used to automatically generate tests that assert functionality.

4.1. Analyzed binaries
Around %10 of the challenges used floating point operations during their work, as it can be seen in Table 1. That was a problem because we rely on BAP as a layer between our tool and the binary code and at the time we were implementing this system, BAP didn’t support floating point operations. A few weeks ago, a new version of BAP was released which supports floating point operations and a stable release should be out this October, 2019, therefore we leave as future work the support of floating point operations by our tool.

Two of the binaries called "EthernalPass" and "middleout" could not be analyzed because they made BAP crash while lifting the code. Although we have not had the opportunity to test them in the latest version of BAP, two binaries is not a significant number compared with our whole dataset (246).

Finally, the repository with the dataset mentions that some of the challenges (13) are not working on Linux yet, meaning some of the functionality tests are not passing. Considering that is out of our hand we ignored those challenges and focused on the remaining ones. All in all, we successfully analyzed 186 out of 246 challenges (76%).

4.2. Found vulnerabilities
All experiments were performed in a virtual machine running Ubuntu 18.04 with 24 cores (Intel Xeon 2.10GHz) and 100GB of RAM.

As it can be seen in Table 2 we have found several types of vulnerabilities in our dataset. We will now further describe the exact vulnerabilities and the reason for a higher or lower number of false positives.

The type of vulnerability that we found more frequently was an Out-of-bounds write. That was also the type of vulnerability where we had the most false positives. To explain why such a high number of false positives, we must explain how we implemented the heap. The inner workings of the heap are tricky to simulate (following the standard implementation [4], one had to hold several free node lists, fast bins, etc) so we made some simplifications.

Unless space was already allocated for the heap, we allocate every address after the binary page and before the next allocated page. "Allocated" in this context just means we save the start and end address of the page and we use it as if it was the heap. Our heap implementation consists of a list of free nodes. Initially, there is only one free node, occupying all allocated space. When the function malloc is called, we take the first sufficiently sized free chunk, save the usual meta-data at the head of that chunk and mark it as allocated.

Those simplifications could however cause problems in some functions that use the heap but those problems can be mitigated by having summaries for those functions, which have knowledge about the way we implemented the heap. We noticed that in most cases where we had a false positive either in Out-of-bounds Write or Heap meta-data corruption.
vulnerabilities it was due to a missing summary for a function that interacts with the heap. This indicates that having summaries simulating only a simplified model of the behaviour of the function that they are supposed to imitate can cause many false positives.

The most accurately detected vulnerabilities are **Stack-based buffer overflows**. Stack-based buffer overflows are detected by keeping a list of elements that are stored in the stack that should not be writable by the user like return addresses, saved base pointer or other saved registers. If they are not modified in the usual way (respectively, the ret, leave and pop instructions in the end of the function) then we report that a buffer overflow has been detected.

We have found 180 vulnerabilities, from which 62 (34.4%) were false positives. Despite having a high number of false positives, the final count of accurately found vulnerabilities in CQE challenges (82) is about the same as the ones detected by the first place team (78). Of course, this comparison is unfair for both sides:

- Some teams were in the funded-track which involved funding before the qualification of up to $750,000 per team, namely CodeJitsu, ForAllSecure, TECHx and four other teams that unlike the first, did not qualify in the CQE.
- Some teams like Shellphish already had a working system to detect vulnerabilities. A basic version of Angr already existed before the qualifications, although it suffered a dramatic improvement after the qualification round.
- Each CRS had a full team working on it and one year to be developed and improved before the qualification round.
- Some teams ran their tools in high end servers during the qualifications, having access to resources that are impossible for us to match. TrailOfBits used 10692 cores and 17820 GB of RAM in the qualification round.
- We had access to the full dataset of the competition while the event teams only had access to a few dozens of CBS before the CQE.
- We do not generate POVs with the usual format, although our system is able to generate exploits in some circumstances and was successfully used in some CTFs already, both in reverse engineering problems and binary analysis.

4.3. Similarity Coefficients comparison

We compared the time it took to perform symbolic execution when using different similarity coefficients to compute the likeliness of a block of instructions to contain a vulnerability. We evaluated our own similarly coefficient and compared it with one used in Staged Program Repair (SPR), and with Ochiai’s. Note that this evaluation was performed on the same server as the previous one and we analyzed one challenge at a time, with a maximum time limit (timeout) of 5 hours and maximum memory usage of 80% of the available memory (0.8 * 100 GB = 80 GB). Each POV was analyzed independently and we generated 5 functional tests automatically for each challenge, using the provided state machines. The results are shown in Table 3 and the metrics we used are:

- **Average time to analyze each challenge** - We see that the use of our dynamic similarity coefficient outperforms the others, being able to analyze a program in 1716 seconds (29 minutes) on average, while the others take almost 4 times more.

- **Average time to analyze each challenge, only counting the successfully analyzed challenges** (no timeouts or memory limit exceeded) - We decided to use this metric because we knew that computing our heuristic is much slower than computing the others due to the possibility of having to compute relaxed paths, which is done by iterating all instructions on the compressed trace that follow the instruction we are currently on. Despite that, our similarity coefficient should allow us to follow the trace much more accurately, thus enabling to analyze more challenges than before. We can see from the results that if we only count with the challenges that we can successfully analyze in time and using a reasonable amount of memory then our heuristics actually performs worse (in terms of time) on average than the other two.

- **Number of timeouts and number of challenges that exceeded memory** - Here we can see the reason why our dynamic similarity coefficient outperforms the others overall but takes more time if we only count with successfully analyzed challenges. The other heuristics not only had 4 times more timeouts but some challenges also crashed due to lack of memory. This shows that using our heuristic we can analyze much more binaries even though we take more time on average to analyze each binary, which results in a higher number of accurately found vulnerabilities.

4.4. Compression Algorithm

We evaluated the compression rate achieved by our compression algorithm described in section 3.7, applying it to 185 CBS in our dataset. We only considered 185 CBS because that was the number of challenges for which we successfully generated a trace (the main cause being Linux incompatibility). The
Table 3: Table describing the analysis results of using our tool with different similarity coefficients

<table>
<thead>
<tr>
<th></th>
<th>Ours</th>
<th>SPR</th>
<th>Ochiai</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average time (seconds)</td>
<td>1506</td>
<td>6156</td>
<td>6532</td>
</tr>
<tr>
<td>Average time successful</td>
<td>217</td>
<td>168</td>
<td>241</td>
</tr>
<tr>
<td>Timeouts</td>
<td>19</td>
<td>80</td>
<td>92</td>
</tr>
<tr>
<td>Memory Limit Exceeded</td>
<td>10</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>Vulnerabilities Found</td>
<td>118</td>
<td>89</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 4: Table describing the values for the compression factor $r$ achieved in each challenge of the dataset

<table>
<thead>
<tr>
<th>Compression Factor</th>
<th>#Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 &lt; r &lt; 1$</td>
<td>0</td>
</tr>
<tr>
<td>$1 &lt; r &lt; 2$</td>
<td>28</td>
</tr>
<tr>
<td>$2 &lt; r &lt; 4$</td>
<td>43</td>
</tr>
<tr>
<td>$4 &lt; r &lt; 8$</td>
<td>39</td>
</tr>
<tr>
<td>$8 &lt; r &lt; 16$</td>
<td>28</td>
</tr>
<tr>
<td>$r \geq 16$</td>
<td>47</td>
</tr>
</tbody>
</table>

| Average Compression Factor | 36.629 |
| Standard Deviation         | 92.460 |
| Minimum                     | 1.000  |
| Maximum                     | 64.568 |
| Average Compression Time    | 39.6s  |

results are presented in Table 4. We achieved an average compression rate of 36.629, meaning that on average, the size of the compressed trace is 37 times smaller than the original size.

The average time to run our compression algorithm for an execution trace is 39.6 seconds, which not only is much smaller than the average time that our tool needs to analyze a vulnerability (1716 seconds) but can also be computed just once for each challenge.

On the other hand, Hamou achieved a compression factor of 1086 and the average time to run their compression algorithm for an execution trace is 39.6 seconds, according to the experiments we made in our dataset. Hamou’s algorithm is clearly better, mostly due to the linear complexity and an efficient use of the cache therefore we decided to compare the similarity coefficients using Hamou’s algorithm to compress the execution traces for our heuristics. The most compressed execution trace was more than 178000 times smaller than the original trace and the longest time it took to run Hamou’s algorithm was 216 seconds. W

5. Conclusions and Future Work

In this thesis we have studied how to design and implement an autonomous system capable of detecting and exploring software flaws with no human interaction whatsoever. We extensively used symbolic execution to iterate over all possible program paths and for each path we applied safety policies and performed dynamic taint analysis so as to detect leaks and other vulnerabilities like format strings. The safety policies allow to detect several kinds of vulnerabilities namely buffer overflows, heap metadata corruption, arbitrary read and write, among others.

We leave as future work the adaptation of our tool, specially the BIL Interpreter to add support for floating points, considering that BAP already does. Safety policies are the heart of our detection system. Adding more safety policies in the future is crucial to allow detection of further and more complex vulnerabilities, which results in a better accuracy (i.e. less false negatives) in the analysis of a software. It is also important to implement more summaries which should allow a faster and smarter execution of code, specially library code.

References

