Thesis

Combining learning with fuzzing for software deobfuscation

Mark Janssen

TU Delft

Thesis

Combining learning with fuzzing for software deobfuscation

by

Mark Janssen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on 20 April 2016 at 14:00.

Student number:	1358316	
Project duration:	August 2015 – April 2016	
Thesis committee:	Prof. dr. ir. J. van den Berg,	TU Delft
	Dr. ir. S.E. Verwer,	TU Delft, supervisor
	Dr. A.E. Zaidman,	TU Delft
	E. Sanfelix,	Riscure

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Software obfuscation is widely applied to prevent reverse engineering of applications. However, to evaluate security and validate behaviour we are interested in analysis such software. In this thesis, we give an overview of available obfuscation techniques as well as methods to undo this effort through reverse engineering and deobfuscation. We research active learning, which can be used to automatically learn state machine models of obfuscated software. These state machine models give insight into the behaviour of the program. We identify opportunities to improve the quality of existing active learning algorithms through the use of fuzzing to generate test cases. We utilise the AFL fuzzer, which uses a genetic algorithm in combination with test case mutation to create test cases for a target program. By using insight into the program's execution for each test case, it can create more relevant test cases compared to implementations that do not use this information. We use the generated test cases to find counterexamples for learned state machine models; these counterexamples can then be used by the learning algorithm to significantly improve the quality of the learned model. Compared to active learning with the W-method for test case generation, our combination of learning and fuzzing learns models of obfuscated programs with up to 343× more states, and consequently incorporates more of the program's behaviour into the learned state machine model.

Preface

This Master of Science thesis is the description of the work I conducted at Riscure and the Cyber Security group at the Faculty of EEMCS at Delft University of Technology. I have been interested in cyber security for many years now, which made the direction for my thesis an easy choice. My graduation internship at Riscure introduced me to the topic of software deobfuscation, which is of great importance when evaluating the security of software products. Combining this topic with machine learning was challenging, but ultimately resulted in the work you now see before you.

I would like to thank a number of people for their support during my project. First of all, I would like to thank my supervisor Sicco Verwer for his guidance and feedback, as well as Eloi Sanfelix and Ileana Buhan for their support at Riscure. I would like to thank Rick Smetsers for co-authoring a paper based on the work in this thesis. I also thank my graduation committee for their comments and remarks. Furthermore, thanks to my colleagues at Riscure for the much needed distraction during my time there. Finally, thanks to my family and friends for all their support over the years of my study.

Mark Janssen
Delft, April 2016

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

In recent years, there has been a rise of interest in techniques to obfuscate software in an effort to safeguard it from prying eyes. By using obfuscation techniques, we can transform a program into a semantically equivalent program that is more difficult to analyse from a reverse engineering standpoint.

There are many use cases for software obfuscation. The broadest use case is the protection of intellectual property (IP) such as protocols or algorithms, where the author does not want their program to be cloned. For example, Skype has used obfuscation to prevent the introduction of Skype-compatible third party programs [28].

Another use case is in digital rights management (DRM) libraries, such as those to prevent software privacy [11] or to limit distribution of video and music: in streaming video and music applications, the content distributor provides the end-user with an encrypted stream that can only be viewed on the distributor's terms [35]. Ideally, the decryption would fully take place in tamper-resistant hardware such that the secret decryption key cannot be accessed by an attacker. However, in practice, distributors make their content available to the widest audience possible, and thus everything is implemented in software. To protect against reverse engineering, the decryption algorithm and its secret keys are obfuscated so that they are hard to recover.

Finally, many malware authors employ obfuscation to make it harder to see how their programs work [61]. This can be to hide a certain vulnerability from being copied by other malware authors, to prevent their botnet command-and-control servers from being discovered, or to conceal what information is collected in the case of spyware.

There are numerous reasons why it would be interesting to reverse this effort. For example, we could be interested in evaluating the effectiveness of the applied obfuscation techniques, in order to assess whether it provides sufficient protection against attacks. In the case of malware, this makes even more sense: if we want to prevent malware attacks and their spread, security researchers need insight into the malware's inner workings [21].

This thesis will focus on the aforementioned topic: given an obfuscated program, how can we get insight into its behaviour? More specifically, we are interested in programs that use external input—

files, network packets or user commands—in their computation. This means that depending on the external environment the program's behaviour changes; we want to know what behaviour the program can exhibit and which input triggers it. A sequence of input and corresponding output of a program can be captured in a *trace*; using several traces it is possible to build a state machine model of the software's behaviour.

To achieve this we will be using machine learning algorithms that can build state machine models from these traces. We can separate two classes of such algorithms: *passive and active learning* methods. In passive learning, existing traces of the target program are used as-is to learn a state machine model. Active learning enhances this method by adding a feedback loop, where the learning algorithm can come up with new series of inputs to be answered, in the form of a new trace, by the target program [2]. Because the algorithm now has control over what traces are used for the learning process, it can potentially create a more complete model.

One critical piece in active learning algorithms is the *equivalence query* to an oracle that can verify whether a hypothesised state machine model is correct, i.e. the hypothesised model is equivalent to the model implemented by our target program. For real-world applications, implementing such an oracle is impossible [43]: after all, we want to learn the model because the model implemented by the target program is unknown. We can, however, approximate the oracle by generating test inputs and verifying whether the output from the hypothesised model is equivalent to the output of the target program (implementing the actual model). When the output differs among the hypothesised and actual models, we have found a *counterexample* and the learning algorithm will need to refine the hypothesis incorporate this new information. Generating the test cases required for this process is a research field of its own; Chow's W-method [14] is commonly used in combination with state machine learning.

Nevertheless, the W-method has some drawbacks: in general a lot of test cases are generated, which results in a rather slow overall learning process. For complex programs with long input sequences, the number of test cases required to validate the hypothesis will explode and make the validation infeasible. We thus want to find new ways to approach the problem of test case generation in the learning context.

The field of *fuzzers* provides some interesting opportunities here. In essence, fuzzers are programs that provide generated inputs (test cases) to a target program and then monitor whether the target program fails, mostly for security problems; e.g. a crash could uncover a exploitable buffer overflow. On one hand, 'dumb' fuzzers use randomly generated inputs that offer no solution to the state explosion problem. On the other hand, more sophisticated fuzzers use mutation-based and generation-based approaches to test case generation, which can yield better results. Mutation-based approaches mutate an existing test case (e.g. with bit flips) to create new ones. Generation-based approaches generate inputs based on a protocol or file format specification, which is effective in terms of the number of valid inputs generated, but also requires one to write this potentially complex specification.

More recently, good results have been achieved by combining mutation-based fuzzing with a genetic (evolutionary) algorithm [64]. This requires a fitness function to evaluate the performance of new test test cases, e.g. a measurement of what code is executed for a certain test case. The fittest test cases can then be used as a source for mutation-based fuzzing, and iterating this process creates an evolutionary approach to fuzzing. We think that using this approach to generate test cases for the equivalence testing will allow for improve state machine learning.

1.1. Research Objectives

Our research objectives can be summarised in a number of research questions to be answered based on the results of our work.

RQ 1 *Can we use learning algorithms to gain insight into the behaviour of obfuscated programs?*

There are many approaches to gaining insight into obfuscated programs, or *deobfuscation* in a generic sense. Various methods are explored in chapter 2. In this work we focus on learning the behaviour of obfuscated programs in the form of a state machine using learning algorithms. Learning algorithms and obfuscation techniques are explained in chapter 2 and in chapter 4 we evaluate how learning algorithms perform on obfuscated programs.

RQ 2 *Can we combine learning algorithms with fuzzing techniques?*

By finding a way to combine learning algorithms with fuzzing techniques we hope to improve the learning process through better test case generation for testing the equivalence of the model. In chapter 3 we describe our exact approach to combining learning and fuzzing.

RQ 3 *Is it possible to use insight in the execution of a program to improve the quality of the learned model?*

In regular learning algorithms, the target program is considered to be a black box. We think there is potential to improve quality of these learning algorithms by using insight into a program's execution, e.g. what sections of code of the target program are executed. In chapter 2 we describe techniques to gain insight into the execution of a program and in chapter 3 we explain our approach to combine this with learning.

To answer our research questions, we need to identify how we can model the behaviour of a program. In addition, we need to explore existing (fuzzing) techniques and (learning) algorithms—along with their implementations—and determine how these can be improved upon. Furthermore, these improvements need to be implemented in a prototype solution. Finally, this prototype solution can be evaluated and compared against other (existing) solutions.

1.2. Outline

This remainder of thesis is organised as follows: in chapter 2, we present a classification of obfuscation techniques along with several publicly available obfuscator implementations. This chapter also covers reverse engineering and deobfuscation techniques, accompanied by an overview of various tools that have been developed for these techniques. Furthermore we give an introduction to active learning algorithms and fuzzing techniques. In chapter 3 we present our approach to combining both in the implementation of a prototype. Subsequently, in chapter 4 we present our results and an evaluation of this approach. We conclude with chapter 5 where we summarise the insights we gained from this work as well as some possible directions for future work.

2

Background and Related Work

Over the years, many techniques and tools have been developed that try to obfuscate software in a way that makes the process of reverse engineering as difficult as possible. More formally, an obfuscator $\mathcal{O}$ takes a program $\mathcal{P}$ and produces a program $\mathcal{O}(\mathcal{P})$ that has the same functionality as $\mathcal{P}$ yet is 'unintelligible' in some sense. While a perfect obfuscator has been proven by Barak et al. [5] to be impossible to create, it is still possible to obfuscate a program in such a way that makes it very time-consuming and thus very expensive to reverse engineer.

2.1. Obfuscation Techniques

In this section we will describe various obfuscation techniques sorted into three major categories. First we will cover control flow obfuscation, followed by data obfuscation, and finally some advanced techniques that combine these two in a hybrid form. We do not cover layout obfuscation techniques that remove useful information, such as identifiers and debugging symbols, because these techniques are straightforward and have already been widely applied in practice for quite some time now.

2.1.1. Control Flow Obfuscation

The control flow of the original language (e.g. C) is usually well preserved in the compiled binary code. For example, constructs like loops (`for`, `while`), conditionals (`if`, `else`, `switch`) and functions are all transformed in a relatively straightforward manner. This also means that code written in a single block (e.g. function, or branch of a conditional) will be located together in binary form as well. Control flow obfuscation alters the flow of control within the code in such a manner that this assumption does not hold any more.

Commonly used control flow based obfuscation techniques are:

Opaque Expressions A widely used obfuscating transformation is the use of *opaque expressions*. The concept is to construct an expression with a known value at obfuscation time, while this value is difficult to figure out for an attacker.

The most common form are *opaque predicates*: boolean valued expressions with a known evaluation value, i.e. it is known whether the expression will return true, false or sometimes true and sometimes false. An example of such a predicate is $(x^2 + x) \mod 2 = 0$, which will evaluate to true for every $x \in \mathbb{Z}$ [16, Sec. 4.3.1].

An interesting additional application of opaque predicates is in software watermarking, as described by Palsberg et al. [41]: each copy of the software is 'personalised' by the author with dynamically generated opaque predicates. If a copy of the software is ever illegally distributed, the author can use the specific combination of opaque predicates to associate the copy with its original recipient.

Computation Transformations There are several different transformations that we can classify as Computation Transformations. First of all, we can insert dead and irrelevant code into the program. This can be easily done through the use of opaque predicates: we can then simply put our generated junk code into a branch that can never be taken. We can also use opaque predicates to extend and complicate loop conditions, without affecting the actual number of times a loop will execute [17].

Aggregation Transformations An important compiler optimisation is the inlining of functions. However, it can also be used to effectively obfuscated the structure of the program: there will be no trace of a function call left. Outlining is the reverse operation, which we can use for obfuscation: we take several sequential statements and extract them into a new function, confusing an attacker with bogus abstractions.

We can also apply variations of this concept, e.g. merging two functions together or cloning functions. Similar approaches can be used on loops, e.g. by (partially) unrolling loops, loop blocking (breaking up the iteration space) or loop fission (creating several loops with the same iteration space) [17].

Ordering Transformations Change code locality: randomise placement of source code blocks (classes, methods/functions) within the binary [16, Sec. 4.1.1]. Within a basic block, data dependency analysis is needed to determine which reorderings are possible.

This transformation can be nicely combined with the previously described inlining and outlining transformation. We can do so by first inlining all code in a function, then randomising the ordering within this function, and finally outlining several contiguous sections of the function, to create bogus functional abstractions with unrelated statements.

Control Flow Indirection By using control flow instructions in creative ways, the actual control flow can be obscured. For example, instead of a regular `jmp` instruction to the target address, a call can be made to a function that then returns to the actual target address [16, Sec. 4.3.5]. Other approaches are also available, such as putting some code in an exception handler and 'calling' that code by triggering an exception.

When focusing on specific architectures, more exotic constructs can be made. Bangert et al. [4] demonstrated "a Turing-complete execution environment driven solely by the IA32 architecture's interrupt handling and memory translation tables, in which the processor is trapped in a series of page faults and double faults, without ever successfully dispatching any instructions.". Another possibility is to exploit instruction set complication, such as done by the Domas [23] using the Turing-complete `mov` instruction in the x86 architecture Dolan [22].

Algorithm 1 A simple implementation of a coin operated vending machine that accepts coins with dominations of 10, 20 and 50. The cost of an item is 50, which can be paid with multiple coins. After receiving an input, the machine prints 1 if it vends an item, and 0 otherwise.

```
int balance = 0
loop
  int coin = scan()
  balance += coin
  if balance ≥ 50 then
    balance -= 50
    print 1
  else
    print 0
  end if
end loop
```

Control Flow Flattening Another commonly used tool is *control flow flattening*, first described by Wang et al. [56], and extended upon by Chow et al. [13]. It removes the control flow structure that functions have, e.g. nesting of loop and conditional statements, by 'flattening' the corresponding control flow graph using *indirect control transfers*. To do this, we place every basic block as a `case` inside a `switch` statement (the 'dispatcher'), and wrap the `switch` inside an infinite loop.

As an example, consider the implementation of a simple vending machine in Algorithm 1. We can draw up a control flow graph of this program (Figure 2.1, left) and modify the code to use a central dispatcher that determines the next code block based on a previously set variable , which adds a layer of indirection (Figure 2.1, right). While the flattened control flow graph is still intelligible, for larger programs with more complex control flow this will not be the case.

These transformations are hard to reverse based on a few basic premises, such as *may alias* analysis (determining which pairs of expressions might refer to the same memory location) being an NP-hard problem [29]. Wang et al. state that, in the presence of general pointers, the problem of determining precise indirect branch target addresses is NP-hard (proven through reduction from the 3-SAT problem). On the other hand, Chow et al. prove that determining the reachability of a flattened program dispatcher is PSPACE-complete.

2.1.2. Data Obfuscation

In addition to control flow based obfuscation, there are several transformations that work based on how data is stored:

Storage and Encoding Transformations Storage and encoding transformations go hand-in-hand; storage transformations attempt to choose unnatural storage classes for both dynamic and static data, whereas encoding transformations attempt to choose unnatural encodings for common data types. [17] An example would be to store integer data with a certain predetermined offset, so that the actual value is not present in memory. For string values, we could use a scrambled version of ASCII so that common decoding tools will not provide read-

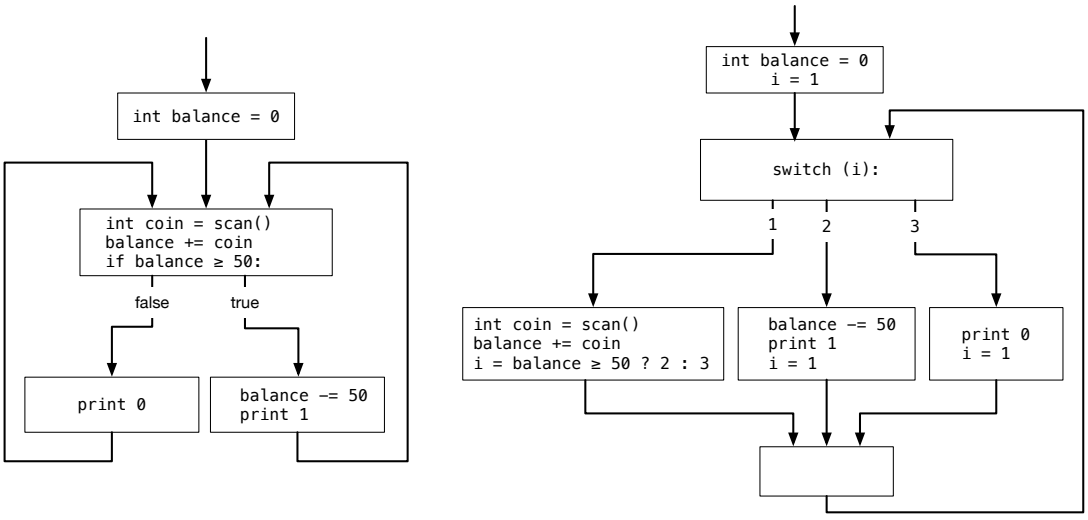

Figure 2.1: An illustration of control flow flattening applied to the program from Algorithm 1. On the left, we show the program's control flow in diagram form. On the right, we have flattened the control flow by adding a layer of indirection: a dispatcher (`switch` statement) at the top decides which block of code is executed based on variable `i` which was set in other code blocks.

able strings. However, to do calculations on these transformed values, we will need to decode them, which makes them easier to figure out for a reverse engineer.

To avoid this, we could use *homomorphic functions* so that we can perform calculations directly using the encoded values [66].

Constant Unfolding In compiler theory, *constant folding* is the process of recognising and evaluating constant expressions at compile time rather than computing them at runtime. However, this makes any constants easy to spot for a reverse engineering. For obfuscation purposes, we can invert this process so that constants are instead represented by an (overly) complicated operation with the constant as a result [19].

Instruction Identities Many instructions can be replaced by (a series of) semantic equivalents that will obscure their meaning. For example, instead of a direct jump (`jmp addr`) we can pretend to return from a function call (`push addr; ret`) with the same effect. Inverting a register, which is usually done directly using `not reg`, can also be done using an *xor* operation: `xor reg, 0xFFFFFFFF` [49]. While these instruction replacements do not impact the behaviour of the program, it makes the disassembly of the program harder to understand.

2.1.3. Hybrid Control Flow and Data Obfuscation

There are several techniques that combine both control flow based obfuscation and data based obfuscation:

Virtualisation Programs are usually compiled to instructions specific to a physical machine architecture (e.g. x86) or virtual machine architecture (e.g. the Java Virtual Machine). However, it is also possible for one to create their own custom instruction set and accompanying

interpreter, which means that a reverse engineer would have to learn the internals of that new instruction set first [16, Sec. 4.1.1.4].

This principle is used by *virtualisation-based obfuscators*, which a use virtual environment and accompanying interpreter to execute bytecode programs. The language accepted by the interpreter is randomly chosen at the time of obfuscation, including the interpreter implementation. The original program can then be converted to this new language, resulting in a semantically equivalent program [44].

However, virtualisation comes at a price in terms of execution speed: the indirection caused by the interpreter causes severe overhead during runtime [17, Sec. 6.2.5].

White-Box Cryptography In modern cryptography, the device performing cryptographic operations is considered a trusted black-box. This means that an attacker only has access to input and output of the operations, and does not have access to the key material used to perform these operations. However, there is a demand to deploy cryptography in situations where such a trusted black-box is not available, such as when deploying software-based DRM without a secure element or other trusted hardware. In these cases, an attacker can trivially extract key material in the *white-box attack context* by inspection of the (binary) code or by using debugging tools.

White-box cryptography tries to address this problem; Wyseur [59] states: "The challenge that white-box cryptography aims to address is to implement a cryptographic algorithm in software in such a way that cryptographic assets remain secure even when subject to white-box attacks." In terms of implementation, the idea is to embed both the fixed key and random data in a composite form, from which it is hard to derive the original key.

Chow et al. have developed both white-box DES and AES implementations based on this idea [11, 12]. Their strategy is to transform the cryptographic algorithm into a series of lookup tables, as illustrated in Figure 2.2. Additional encoding techniques can be applied to each of the lookup tables to strengthen the obfuscation. However, these implementations have later been broken by Goubin et al. [24] and Wyseur et al. [60] respectively, using cryptanalysis techniques to recover the key. More recently, Bos et al. [8] introduced a new attack that exploits the characteristics of memory operations in cryptography algorithms to recover keys from white-box cryptography implementations using nothing more than a number of software execution traces.

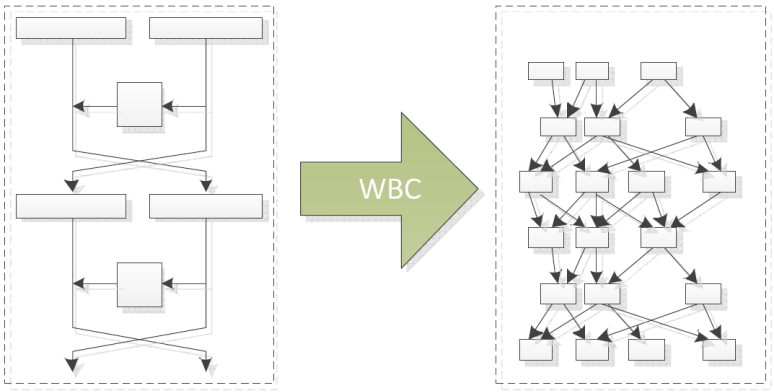

Figure 2.2: In white-box cryptography, one technique is the use of lookup tables. The cryptography operations, represented as boxes in the original implementation (left), are replaced with a series of lookup tables (right); the arrows in the diagrams represent the data flow through the algorithm. The secret key is hard-coded into the lookup tables and is protected by randomisation techniques that are applied. [59, Figure 2]

2.2. Obfuscator Implementations

There are many commercial obfuscator implementations available, for which the exact working is usually not published because of general 'security through obscurity'-underpinning of obfuscation. Some of these commercial products include VMProtect [55] that produces virtualised code, the Morpher obfuscation service [39], Arxan Application Protection [3], and whiteCryption [58]. The last two vendors also provide white-box cryptography solutions. Furthermore, there are many proof-of-concept obfuscators developed in academic research that are not released as a full-featured project, and are thus not yet for any production use.

Fortunately, as of recently, there has been increased activity in work on freely available obfuscator implementations:

Obfuscator-LLVM Over the last ten years, the LLVM compiler infrastructure project [33] has seen increased attention from developers and companies around the world. The LLVM project includes the *LLVM Intermediate Representation (IR)* assembly language. It is used as a output language for frontends (e.g. the Clang C compiler), as well as an input to backends that emit binary code suitable for running on actual hardware (e.g. the LLVM ARM Backend). Between the frontend and backend, a generic optimiser can be used to perform well-known compiler optimisations directly on LLVM IR code. This means that this component can be used for any combination of frontends and backends. The LLVM project includes the *LLVM Optimizer* that achieves exactly this.

In the same spirit, one could develop an obfuscator that works on LLVM IR code. This idea has been pioneered by Souchet [53] and Chen et al. [10]. Later, the *Obfuscator-LLVM* open source project was developed by Junod et al. [31]. According to the authors it "[...] supports basic instruction substitutions, insertion of bogus control-flow constructs mixed with opaque predicates, control-flow flattening, procedures merging as well as a code tamper-proofing algorithm embedding code and data checksums directly in the control-flow flattening mechanism."

Tigress Originally developed by Collberg et al. [15] at the University of Arizona, Tigress is 'diversifying virtualizer/obfuscator for the C language'. While the project is not open source, pre-compiled binaries are available free of charge. It performs source-to-source transformations, i.e. works directly on the C source code instead of an intermediate representation.

The authors claim that "[...] Tigress protects against static de-virtualization by generating virtual instruction sets of arbitrary complexity and diversity, by producing interpreters with multiple types of instruction dispatch, and by inserting code for anti alias analysis. Tigress protects against dynamic de-virtualization by merging the real code with bogus functions, by inserting implicit flow, and by creating slowly-executing reenetrant interpreters. Tigress implements its own version of code packing through the use of runtime code generation." and that "Tigress' design is similar to that of commercial tools, such as Cloakware/IRDETO's C/C++ Transcoder."

In the next chapter, we will cover advances in deobfuscation techniques to counter these obfuscation efforts.

Even though more advanced obfuscation techniques are being developed and deployed continuously, there is still a rat-race to beat these through new reverse engineering and deobfuscation techniques. In the next section, we will cover various advances in reverse engineering in general and deobfuscation in particular: first we will describe various techniques, followed by an overview of the numerous tools available for this purpose.

2.3. Reverse Engineering and Deobfuscation Techniques

In this section, we will describe a number of static analysis techniques, some dynamic techniques and finally some approaches that combine static and dynamic analysis in a hybrid approach.

2.3.1. Static analysis techniques

Static analysis techniques use only knowledge that can be extracted from the binary code of the target program, that is, without executing it on a (virtual) machine. These include:

Constants identification and pattern matching A relatively simple static analysis technique is finding known patterns in the code. For example, if the target program uses a cryptographic primitive like AES, DES or SHA-1, we can try and detect this. We could match a x86 assembly code snippet, some 'magic' constants of the algorithm, structures like S-boxes, or the string for an import of a cryptographic function call [25].

It is also possible to use pattern matching techniques to find duplicated code blocks. By replacing these duplicated blocks by a function call, the effect of function inlining can be undone, which leads to more understandable code. A similar approach can be used for opaque expressions and constant unfolding: once an expression identified and evaluated, all of its occurrences can be replaced by the evaluated value.

Symbolic execution Originally introduced by Boyer et al. [9], symbolic execution is a way to analyse what set of inputs to a program lead to every possible execution path. This is done by instrumenting the program code and assigning symbolic values to every variable that depends on external input. When a branch is then encountered, a constraint solver can be used to solve

the branch expression in terms of these input symbols. It is thus possible to determine what inputs are needed to reach any specific branch in the program.

However, symbolic execution has severe limitations. For example, if there are many possible paths in the program, it is possible to encounter a so-called *state explosion* or even constraints that are too complex to be solved in reasonable time. This makes symbolic analysis infeasible for many real-world programs, and so we have seen a move to combining symbolic execution with dynamic analysis techniques in *concolic execution*, which we will cover later.

Decompiling Decompilers such as the *Hex-Rays Decompiler* [46] can produce C-like pseudocode from x86, x64, and ARM binary code. In comparison to the low-level assembly code generated by disassemblers, the high-level pseudocode is more readable. However, the value of these decompilers dramatically decreases for most obfuscated code, since the obfuscated code is not structured in a way that is nicely represented in decompiled (pseudo)code.

Compiler optimisation If a program's bytecode can be reliably disassembled to an intermediate representation (such as LLVM IR) then we could try to apply optimisations to this IR code. This approach was used by Rolles [44] to circumvent protection by the VMProtect virtualisation-based obfuscator. A similar approach was used by Guillot and Gazet [26]; however, these authors note that dissembling to IR code is not always possible. They also propose the use of compiler optimisation projects like LLVM instead of their own, and propose extending this technique with concolic execution approach.

2.3.2. Dynamic analysis techniques

In contrast to static analysis, dynamic analysis techniques execute or emulate the program and instrument it in a way that gives us insight into its operations. Some commonly applied techniques are:

Dynamic tracing Whereas symbolic execution tries to capture all execution paths, dynamic tracing techniques focus on capturing a full trace of one concrete execution of a program. This allows for offline analysis and visualisation of the instructions executed, memory locations accessed, API or system calls made, and so forth. Because this trace only covers a single execution, all conditional branches will have a concrete path taken and thus any unreachable code (e.g. because of opaque branch expressions) will not be visible in the trace. However, this also means that the trace might not include some conditionally reachable code, e.g. code that depends on input to the program (if any).

Data slicing and tainting First introduced by Weiser [57], data slicing can be used to determine which operations (instructions) and other data (variables) were used to calculate a certain piece of data.

Taint analysis, as described by Schwartz et al. [48], works the other way around: it is used to track the flow of data from a certain source (which we mark as *tainted*) to a sink. Every operation that uses the tainted data will have its outputs also marked as tainted, so that we can follow the full flow of our input data and how it affects output.

Fuzzers To test boundary cases in handling the input given to a program, *fuzzing* can be used to generate a large amount of test cases. As described by Oehlert [40], the goal is to generate a large amount of semi-valid input data, i.e. data that is accepted by the program as valid

input but that might trigger unintended consequences when processing it. The target program is then run with this generated input data, and any test cases that cause the program to unexpectedly fail (crash) are saved for later inspection. Because this process can be executed in an automated manner, it is easy to assess millions of test cases: something that would not be practical when manually writing these test cases. Furthermore, the process is entirely black-box: the code or internal state of the program is not inspected at all.

There are two ways to generate test cases: *mutation-based* and *generation-based*. Mutation-based fuzzing involves taking a given, known-good test case and mutating that into new test cases, e.g. using bit flips. A generation-based fuzzer uses a format specification to generate test cases according to that specification, e.g. with fields of specific lengths.

On the one hand, a mutation-based fuzzer has no assumptions about the format or application and might reach trigger test cases that the program's author has never considered, such as bugs in handling invalid inputs that can occur in parser code). On the other hand, a generation-based fuzzer requires one to create a (potentially non-trivial) format specification for the test cases. However, because this specification is created with knowledge of the target program, this fuzzing technique has more information and can potentially create more advance test cases to be able to advance further into the target's code paths.

Unfortunately, there are several common problems with fuzzing. For example, a fuzzer will need to understand hashes and checksums in protocols, or the target program will simply reject the input entirely. For digitally signed, encrypted or compressed input data, the fuzzer needs to apply this transformation after the raw test case data is generated.

Because a fuzzer might produce many test cases with the same result, it is useful to reduce or minimise test cases to the shortest possible input that produces the same unexpected result in the target program.

State machine learning A completely different approach was used by Aarts et al. [1] to automatically learn state machines from a system in a black-box fashion. The target program or system (system under test, SUT) can modelled as a deterministic finite automaton (DFA) or Mealy machine that is learnt by a learning process, without any knowledge of the SUT internals.

The principle behind these learning algorithms is to take *traces* of input and output behaviour of a target program and infer a state machine of the program's behaviour from them. When this is done using existing traces we can speak of *passive learning*; when we allow the learning algorithm to query the target program (ask for its output given some input) so as to obtain additional traces we speak of *active learning*.

An example passive learning implementation is is DFASAT by Heule and Verwer [27]. Active learning algorithms and implementations are covered in more detail in section 2.4.

2.3.3. Hybrid static and dynamic techniques

Hybrid approaches combine both static and dynamic techniques to obtain a more complete picture of the target program's operations. Such techniques include:

Advanced fuzzers An improvement over regular fuzzers is to use code instrumentation (either compile-time or runtime) to validate whether the fuzzer makes any progress in terms of dis-

covering new code paths. In the *American Fuzzy Lop (AFL)* fuzzer, created by Zalewski [65], code instrumentation is combined with a genetic algorithm to discover new test inputs that trigger new edges in the program control flow. This strategy allows the fuzzer to discover test cases for file formats and other data structures without intervention.

AFL is described into more detail in section 2.5.

Concolic execution By combining concrete and symbolic execution into a hybrid strategy, we can leverage the advantages of both. For this technique, the portmanteau term *concolic execution* was coined. It was pioneered by Sen et al. [51] in 2005, in the *CUTE* concolic unit testing engine for C.

In concolic testing, test input generation is automated by combining concrete and symbolic execution of code. Whereas more traditional techniques use either concrete execution or symbolic execution to build constraints to generate concrete test inputs from, concolic testing tightly couples both concrete and symbolic execution and runs them simultaneously so that each gets feedback from the other Sen [50]. To solve the constraints and build test inputs, SMT (satisfiability modulo theories) solvers are used.

While originally developed for unit testing, we can use the same techniques to explore all reachable paths in an obfuscated program. Where conditions on branches might be hard to figure out for a human analyst, SMT solvers can quickly determine valid inputs (if any).

One of the newer frameworks to uses concolic execution is *angr*, developed by researchers in the Computer Security Lab at UC Santa Barbara. It was originally developed under the name *Firmalice*, a tool that was used to automatically detect authentication bypass vulnerabilities in binary firmware [52].

The framework focuses on both static and concolic analysis, and consists of several subprojects that could be reused individually: a binary code loader, a simulation engine, an abstraction for constraint solving, and others. These subprojects are all used in the angr suite itself, for which a GUI ('angr-management') is also available. Internally, the Z3 constraint solving library [20] is used. Although angr is in active development, it is currently not yet suitable for analysis of (heavily) obfuscated software. Furthermore, it requires a lot of manual programming and configuration work, which may not be reasonable when analysing a large target program.

Statistical analysis of I/O Scrinzi [49] describes methods to use analysis of program I/O, i.e. its memory access locations and contents, to recognise cryptographic operations within programs, even heavily obfuscated ones. Bos et al. [8] and Sanfelix et al. [47] expand upon this methods to extract key material from common white-box cryptography solutions, while requiring limited knowledge of the white-box implementation's internal structure or encoding parameters.

2.4. Active Learning

As mentioned earlier, active learning of state machine models can provide us with insight into the behaviour of a target program. In this chapter we give a more in-depth explanation of active learning, state machines in general, available algorithms and implementations as well as some related work.

2.4.1. Introduction to Active Learning

Active learning is a machine learning method that uses a semi-supervised learning process. This means that a learning algorithm is able to interactively query an expert source (commonly referred to as the 'teacher': the user, or some other information source) to obtain desired outputs for new data points. Contrary to passive learning methods, where all outputs for learning are obtained without reference to the learning algorithm, the expectation is that active learning methods require substantially fewer outputs to learn a correct and complete model.

One prominent approach to active learning was described by Angluin [2]. In this work, she proved that one can efficiently learn a finite state machine that describes the behaviour of a system if a teacher is available to answer two types of questions about the (to the learner unknown) *target* finite state machine:

- In a *membership query* (MQ) the learner asks for the target's output in response to a sequence of inputs. The learner uses the outputs for a set of such queries to construct a *hypothesis* finite state machine.
- In an *equivalence query* (EQ) the learner asks if its hypothesis is equivalent to the state machine of the system. If this is not the case, the teacher provides a *counterexample*, which is an input sequence that distinguishes the hypothesis and the system. The learner then uses this counterexample to refine its hypothesis through a series of membership queries.

This process iterates until the learner's hypothesis is equivalent to the target finite state machine.

To apply this learning process in a reverse engineering context, we first need to establish a way to model the behaviour of a program.

2.4.2. Mealy machines

A large class of programs have deterministic behaviour solely affected by what input is provided, e.g. by the user or through a network interface. We can model such programs as a *Mealy machine* [38], a finite-state machine whose output is determined by both the current state and the current input. More formally, a Mealy machine is a 6-tuple $M = (Q, Q_0, I, O, \delta, \lambda)$ consisting of:

- a finite set of states Q;
- an initial state Q_0;
- an input alphabet I, a finite set of input symbols;
- an output alphabet O, a finite set of output symbols;
- a transition function $\delta : Q \times I \to Q$ that maps pairs of a state and an input symbol to the subsequent state;

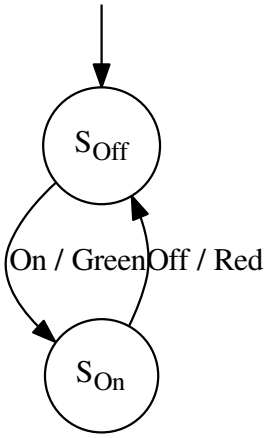

Figure 2.3: Visualisation of a simple system modelled as a two-state Mealy machine

- an output function $\lambda : Q \times I \to O$ that maps pairs of a state and an output symbol to an output symbol.

As an example, consider a simple system with an *On* and an *Off* button; the former will turn an LED green and the latter will turn that same LED red. We can model this system as a Mealy machine consisting of two states S_{On} and S_{Off}. The initial state is S_{Off} because our system is turned off by default. The input alphabet contains two symbols, one for each button, *On* and *Off*. The output alphabet is the state of the LED: *Red* and *Green*. The transition function and output functions can be represented in a table that shows the new state and output for each combination of the current state and an input symbol. Here, we choose to directly visualise the model as a graph, as shown in Figure 2.3.

The key insight to applying active learning in the context of software obfuscation is that obfuscating a program's code will not change its external behaviour, i.e. its Mealy machine model is equivalent. The automated learning of such a state machine for a given target can be done with active learning algorithms.

2.4.3. Algorithms

Generally speaking, two flavours of Angluin-style for actively learning state machines exist. The first branch of algorithms are based on Angluin's original L* algorithm, which we will describe here. The second branch of algorithms are based on the work of Kearns and Vazirani [32]. From this branch of work we describe the more recent *TTT* algorithm by Isberner et al. [30], because it is particularly well-suited to handle long counterexamples.

Angluin's L* algorithm incrementally constructs and refines a so-called *observation table*. The rows and columns of an observation table are labelled by prefixes and suffixes of membership queries respectively, and the cells are filled with the last output in response to this membership query. The learner asks membership queries to its target to fill this table.

For this hypothesis, an equivalence query is performed. This query can return a counterexample if a test case is found that is not adequately handled by the hypothesis. In Angluin's original description, the counterexample and all of its prefixes are added to the observation table, and new membership queries are required to complete the table. This procedure iterates until no counterexample can be found any more. At this point, it is assumed that the learned state machine is equivalent to the actual state machine of our target.

A problem with the L* algorithm is that it is not prepared to deal with long counterexamples, as these add a lot of (possibly) redundant information to the observation table. The distinguishing characteristic of the TTT algorithm [30] is its redundancy-free storage of observations. By thorough analysis of counterexamples, only essential refining information is extracted and stored. The algorithm is therefore particularly well-suited for runtime verification contexts where counterexamples, obtained through e.g. monitoring, can be excessively long.

As mentioned in our introduction to active learning, we need to define a method of answering equivalence queries as part of the learning process. This is a requirement for any of the algorithms discussed above. The most widely studied approach for this purpose is *conformance testing*.

One of the main advantages of using conformance testing is that it can discover all counterexamples for a hypothesis under the assumption that we have an upper bound n on the number of states in the target state machine. An example of such a conformance testing method is Chow's W-method [14].

Unfortunately, conformance testing has some notable drawbacks that hamper its applicability in state machine learning. In practice, it is very hard to determine this upper bound. Furthermore, it is known that testing becomes exponentially more expensive for higher values of n [54]. This motivates the search for alternative techniques for implementing equivalence queries.

2.4.4. Implementations

There are a number of libraries that implement the discussed learning algorithms. We will consider a few of them:

LearnLib LearnLib [43] is a commonly used library for learning state machines. It is an open-source Java library with implementations of multiple commonly used learning algorithms. Recent work that makes use of active learning has used LearnLib; we will discuss this in subsection 2.4.5. The library has seen active development in recent years.

libalf 'The Automata Learning Framework' [7], or *libalf* for short, is another implementation of L* and other algorithms. However, it has not been used in recent papers, and the latest release of the library was in 2011.

Because of its recent active development, we will rely on the LearnLib for the implementation of learning algorithms. In particular, we will use the 'classic' L* algorithm, with room for change if needed, e.g. when the number of states for a particular target program is so large that another algorithm (such as TTT) would be more suitable.

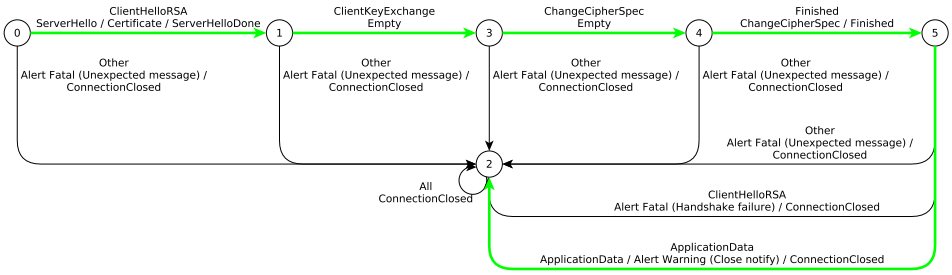

Figure 2.4: Learned state machine model for RSA BSAFE for Java 6.1.1 [45, Figure 6].

2.4.5. Related work

Learning models for TLS implementations The same combination of the L* algorithm and LearnLib implementation was used by Ruiter and Poll [45] to learn a model of protocol behaviour of various Transport Layer Security (TLS) implementations. TLS is widely used to secure network connections, for example in HTTPS. By learning a model of how the implementations work, logic errors can be discovered through manual inspection and/or formal verification of the learned model. An example of such a logic error would be that a TLS connection can be established without authentication of the other party (server) through its certificate. The learning process was conducted in a black-box fashion, i.e. without any knowledge of the internals of the TLS implementation. One of the learned models, of the RSA BSAFE library, is shown in Figure 2.4.

A challenge here is that TLS is a relatively complex protocol that requires state to be kept on both the server and client sides, e.g. to store cryptographic key material from the TLS handshake. This means that to learn a model of an existing TLS server, the learner— which then has the role of the TLS client—must keep appropriate state. This state also affects the content of the protocol messages, e.g. to encrypt messages using the acquired cryptographic key material. This was solved by introducing a *test harness* that functions as bridge between the learner and the target TLS server. It keeps track of state and ensures this state is properly reflected into protocol messages. The learner thus only learns what kind of protocol messages are sent (e.g. *ServerHello*) and not its contents (i.e. not including key material or nonces).

By evaluating the learned state machines, the authors found actual vulnerability in the Oracle Java Secure Socket Extension (albeit at the same time as Beurdouche et al. [6]). The vulnerability meant that, by skipping some messages from the handshake sequence, it was possible to send unencrypted text (plaintext) to the server which was then accepted as if it was encrypted. This would allow man-in-the-middle attacks to take place on one direction of the two-way communication channel.

In general, the behaviour of any system with (finite) deterministic input/output behaviour can be modelled by constructing such a test harness around it. Of course, this test harness could potentially become as complex as the target system, so in the end learning a complete model might not be feasible for every system.

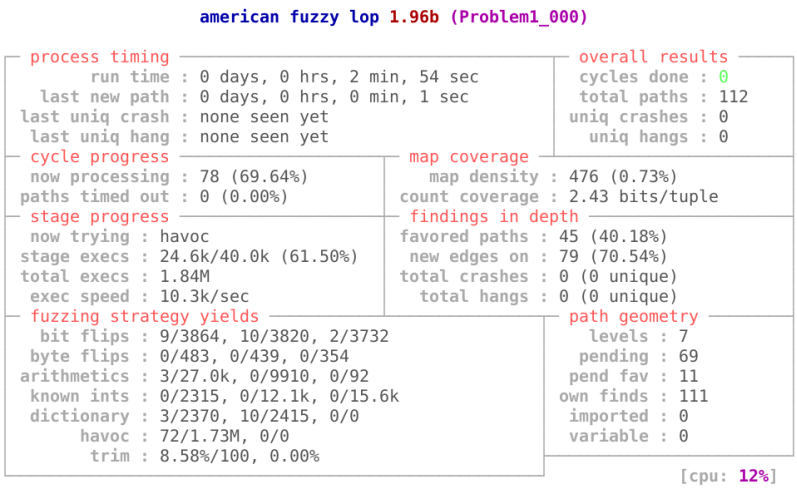

Figure 2.5: Interface of the AFL fuzzer status screen as shown in interactive mode (colours are inverted). The status screen shows information about fuzzing progress: the number of discovered test cases, the effectiveness of used mutation strategies and other statistics.

2.5. Fuzzing

As previously mentioned, the AFL (*American Fuzzy Lop*) fuzzer is interesting for its approach in combining mutation-based test case generation with code coverage metrics. In this section, we give more insight into the techniques behind it.

2.5.1. Overview of AFL

The AFL fuzzer, created by Michał Zalewski, is a mutation-based fuzzer coupled with a simple but solid instrumentation-guided genetic algorithm. By using a form of program control flow edge coverage, it can pick up changes in the program execution to generate better test cases.

AFL functions as a command-line tool, which requires an input directory with one or more initial test cases, an output directory where generated test cases are stored and finally the location of the target program. In interactive mode, a status screen is presented that shows progress information, of which as screenshot can be seen in Figure 2.5.

A high-level, simplified overview on how AFL works is described in Algorithm 2. In rest of this section, we will describe the main components of this flow in more detail.

2.5.2. Measuring coverage

If a mutated test case results in new state transitions in the target program, the test case is seen as valuable. The intuition behind this is simple: we want to cover as much of the target program's code as possible, which gives us the highest chance of discovering all behaviour of the program.

In order to this coverage, AFL uses either compile-time or runtime instrumentation of the control flow (branches, jumps, etc.) of the target program. In Figure 2.6 we have illustrated the control

Algorithm 2 High-level, simplified overview of how the American Fuzzy Lop (AFL) fuzzer works [65]. The fuzzer works on a target program and requires on initial test case as a starting point.

```
load initial test cases into the queue
loop
  take next test case from the queue
  for all available mutation strategies do
    mutate test case
    run target program on test case
    if new state transitions in trace bitmap then
      add new test case to the queue
    end if
  end for
end loop
```

flow of our vending machine example (Algorithm 1). AFL instruments this control flow, and can identify which parts are used when evaluating a test case. Using this knowledge, AFL can decide which test cases cover behaviour not previously seen in other test cases, simply by comparing the result of the instrumentation. For example, using the visualisation on the right side of Figure 2.6, we see that the test case of three coins (2×10 and 1×20) covers all edges of the program control flow, and thus triggers all behaviour.

Internally, the trace bitmap is a 64 kB array of memory shared between the fuzzer and the instrumented target. This array is updated every time an edge in the control flow is taken, as described in Listing 2.1. Note that because the size of the bitmap is finite and the values that represent locations in the code are random, the bitmap is probabilistic: there is a change that collisions will occur. This is especially the case when the bitmap fills up, which can happen when fuzzing large programs with many edges in their control flow. AFL can detect this situation, which can then be resolved by applying instrumentation on fewer edges in the target or increasing the size of the bitmap.

As previously mentioned, AFL can use either compile-time or runtime instrumentation. Compile-time instrumentation has the best performance, and is supported for C and C++ as well as Python, Rust and OCaml with third-party integrations. However, because this requires recompilation of the code, we need the source code of the target program to be available. When the source code is not available, we can use runtime instrumentation which uses emulation (QEMU or Intel Pin) to achieve the result. However, runtime instrumentation is 2-5x slower than compile-time instrumentation [65].

2.5.3. Mutation strategies

At the core of AFL is its 'engine' to generate new test cases. As mentioned earlier, AFL uses a collection of techniques to mutate existing test cases into new ones, starting with basic deterministic techniques and progressing onto more complex ones. The author of AFL, Zalewski [62], has described the main stages that are used:

Walking bit flips This involves performing sequential, ordered bit flips. First, one bit at a time, then two bits and later four bits in a row.

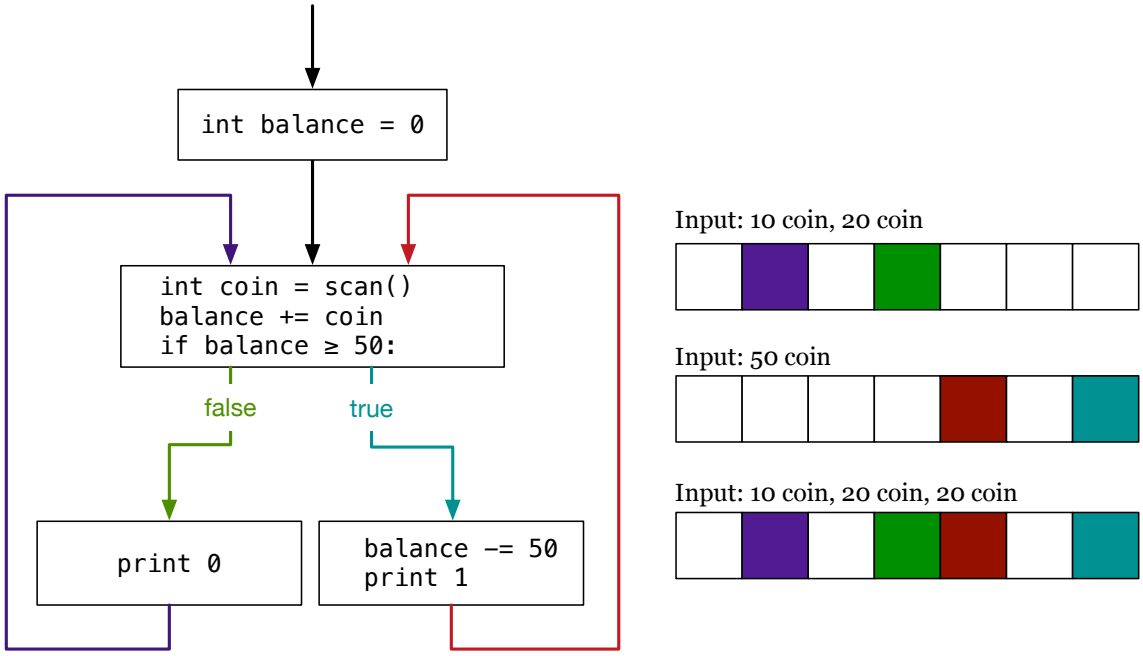

Figure 2.6: A diagram of control flow of the target program (left) and a visual intuition of the AFL instrumentation using several test cases (right). Edges in the control flow (branches in the code) that AFL instruments are represented as coloured that edges in the diagram. When a test case is execution, this instrumentation uses a bitmap to keep track of which edges are used by a certain test case. A visual representation of this bitmap shown on the right: every position in the bitmap represents an edge in the control flow graph, which is either taken (coloured) or not taken (uncoloured).

Listing 2.1: The code that updates the AFL bitmap for every instrumented edge in the program control flow. Every location is represented by a compile-time random value. When an edge in the control flow is taken, the bitmap is updated at in the position of the current location and an xor of the previous location value. This causes every edge in the control flow to be represented by a different byte in the bitmap.

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

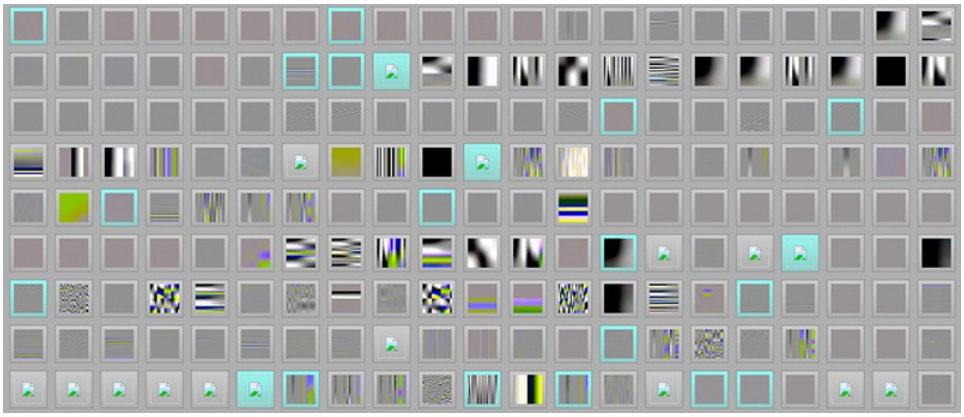

Figure 2.7: A number of test cases generated by fuzzing a JPEG library with AFL. The fuzzer was seeded with only the string "hello". After a one or two days, the fuzzer came up with a blank grayscale JPEG image, 3 pixels wide and 784 pixels tall. This, and subsequent other images that were generated are shown in this figure [64].

Walking byte flips An extension of bit flips is to flip one, two or four bytes at a time.

Simple arithmetic Existing integer values in the test case are incremented or decremented in a range of $[-35, +35]$, which was experimentally chosen by the author.

Known integers Any existing integer values can also be overwritten by values from a set of pre-set integers like -1, 1024 and `MAX_INT`. These values are known to trigger edge conditions in many programs.

Stacked tweaks When the deterministic strategies (above) are exhausted, randomised operations can be applied. This stage consists of a stacked sequence of single-bit flips, setting discovered byte values, addition and subtraction, inserting new random single-byte sets, deletion of blocks, duplication of blocks through overwrite or insertion, and zeroing blocks.

Test case splicing The last-resort strategy involves taking two distinct input files from the queue that cover different code paths and splicing them in a random location. This test case is then passed through the "stacked tweaks" algorithm.

This strategy, combined with AFL's coverage instrumentation, has been found to generate some interesting test cases. One example is the creation a valid JPEG images with only the string "hello" used as an initial test case, as shown in Figure 2.7.

2.5.4. Forkserver

AFL also includes some other tricks to speed up fuzzing, such its forkserver. Fuzzing generally requires a lot of invocations of the target process. Instead of starting a new process for every test case, the AFL forkserver initialises the target process only once, and then forks (clones) it to create a new instance.

On modern operating systems, a process fork is done in a copy-on-write fashion, which means that any memory allocated by the process is only copied when it is modified by the new instance. This eliminates a lot of (slow) memory operations compared to a regular process start [63].

3

Combining Learning and Fuzzing

In this chapter we describe our approach to combining learning algorithms with fuzzing techniques. We will develop of a new prototype tool that uses LearnLib to learn a model based on the input and output of a target process, which we cover in section 3.1. We then explain our approach to combining active learning and fuzzing, implemented by extending the prototype tool with fuzzing techniques.

3.1. Prototype Tool

We created a prototype learning tool in Java, straightforwardly called *Activelearner*, that allows us to perform active learning experiments using LearnLib. Because we LearnLib provides us with implementations of learning and equivalence testing algorithms, the prototype tool can delegate most of the work to classes in LearnLib. We do have to provide a *SUL* (Subject Under Learning), which we can do by creating a class that runs an external target process (feeding it input from the learning/testing algorithm and reading back its output). All other implementation work is in initialising all LearnLib classes and then starting the experiment, which yields a learned, hypothesised model for our target program.

By using the Spring Framework [42] we can make the tool easily extensible with new learning algorithms and equivalence testing algorithms, as well as configure the tool to function with a new target program and associated parameters (e.g. the input alphabet). This easy configuration is done through the runtime dependency injection features of the Spring Framework. During development we define sets of classes, e.g. `lstar` for the implementation of the L* algorithm, that can be selected at runtime using a configuration file or command-line argument. All other parameters (input alphabet, target program, etc.) are set in the same way.

This tool forms the basis of the rest of our work, with e.g. the implementation of fuzzing added later. The full source code of our prototype tool, including the implementation of the combination with fuzzing as described in the next section, is available on GitHub[1].

[1] `https://github.com/praseodym/learning-fuzzing`

3.2. Combining Active Learning with Fuzzing

Several opportunities lie in the combination of (active) learning algorithms with fuzzing techniques. First of all, generating test cases for a given target is a complex problem. We want to test as many code paths as possible while still keeping the total number of test cases low, because we want to ensure that only few test cases cover the same code paths (and thus the same behaviour) for performance reasons.

A 'brute-force' approach where a Cartesian product of the input alphabet (up to a certain length) only satisfies one of these requirements: while we can reasonably expect these test cases to cover all reachable code paths (because the full set of inputs is covered), the number of generated test cases quickly explodes once the input alphabet size increases or the maximum test case length increases. Other methods, like Chow's W-method, are a little smarter but still very much like a brute-force approach (as described in subsection 2.4.3). Also, these algorithms are not good at determining whether a test case is relevant to the target program's implementation.

This is where AFL's approach of using code (branch) coverage for determining test case relevancy comes into play. By using AFL's fuzzing algorithm for test case generation and evaluation, we can generate with a relatively small set of test cases that are highly relevant to the target program. Moreover, the AFL fuzzing process can be run separately from the learning process, and (in contrast to learning using LearnLib) can be run across multiple processor cores or even multiple machines. This vastly speeds up the process of generating test cases for a certain target. The core idea is to use this set of test cases to find counterexamples in the equivalence testing phase of the active learning process.

We can use more components of AFL to improve the learning process. There is quite some overhead in this process, because any external process is restarted for every reset, since most programs do not have some command to fully reset their internal state. By using on the AFL forkserver, we can reduce this overhead. Instead of restarting the process on every reset, we can simply signal the forkserver to have it start a new copy of the process. Another advantage of using the forkserver is that it has good handling of process hangs: whenever the target process execution takes longer than a predetermined duration, either configured manually or by a calibrated run time from running the target a couple of times, the process is terminated automatically. Furthermore, AFL evaluates any new test cases it has ran by looking at its control flow coverage (as described in subsection 2.5.2). Any interesting test cases are saved and can be used as new test cases for AFL's mutation algorithm. This means that interesting membership queries generated by the learning algorithm can be used to fuzz more test cases.

Fuzzing can also be used as a way to discover the input alphabet as accepted by the target program, by looking at the symbols used in the test cases generated by the fuzzer. This input alphabet is required for active learning algorithms; for many target programs it is already known, but fuzzing can help when it is not (or suspected to be incomplete)

In the next section we will describe our implementation of these improvements, and in the next chapter (chapter 4) we will evaluate our approach by comparing it to the performance of existing techniques on several target programs.

3.2.1. Implementation

LearnLib and our prototype active learning tool are both written in Java, whereas AFL is written in C. This means that we we cannot just link the two components together; we will thus need to bridge all communication between the two. For this we rely upon the Java Native Interface (JNI) programming interface that is part of the Java language. It allows managed code running in the Java Virtual Machine (i.e. Java code) to interface with platform-specific native binaries (e.g. external libraries). However, AFL is written as a standalone program, whereas we require a library with main functionality exposed as JNI functions. To achieve this, we need to write the bridging code ourselves. Firstly, to allow the code to be used in a library, we had to rewrite the AFL initialisation code to not rely upon command-line arguments nor automatically start the fuzzing process, i.e. all logic from its `main` function had to be moved to separate initialisation function that can be invoked through JNI. Secondly, all functionality required by our prototype tool has to be made available in JNI functions. These include the ability to run the target program with a given test case, getting the number of test cases newly discovered by the fuzzer, as well as a function that returns the trace bitmap created from the previous target run. The Java code of the JNI interface is listed in Listing 3.1. The overall architecture is shown in diagram form in Figure 3.1.

A design decision in AFL is that it does not care about the target program's output, instead only edge coverage data is used as a measure for test case relevancy. This decision makes it well suited for fuzzing libraries without knowledge of the specifics of the output data (e.g. an in-memory bitmap for an image parsing library), but our learning algorithms rely on output behaviour. We thus extended the AFL fuzzing code to always save data from the target's `stdout` into a shared memory buffer (shared between our library and the forkserver process). The content of this shared memory buffer is returned to the Java component after a successful target run.

We also noticed unexpected timeouts of our target program during learning. These timeouts were transient: retrying execution with the same input would usually succeed, and thus the timeout was most likely caused by resource contention (e.g. CPU usage by other processes on the same machine). Because the learning process cannot continue without the output for the given input, we circumvent transient timeouts by retrying execution (up to a certain maximum number of times).

Another complication is that AFL runs the target program in a non-interactive manner, i.e. it provides the program with input once and then expects it to terminate (resetting all state). This

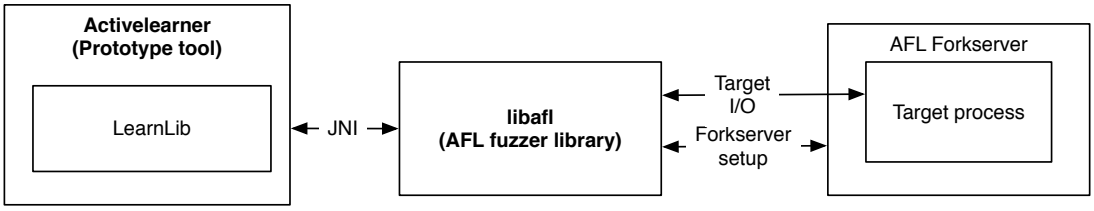

Figure 3.1: Architecture of our prototype learning and fuzzing tool. On the left is our prototype learning tool, which runs in the JVM (Java Virtual Machine). It leverages the LearnLib library for the actual learning algorithms. In the centre is our library version of the AFL fuzzer, *libafl*, to which the learner tool communicates over a JNI (Java Native Interface) bridge. The AFL library controls with the AFL forkserver process, with embeds the target process. Through the forkserver, the AFL library can communicate with the target by sending input (test cases) and receiving the corresponding target output.

Listing 3.1: The class that functions as the Java Native Interface (JNI) bridge to our AFL fuzzer library (libafl), as used within the prototype learner tool. The implementation of these methods is written in C, and is part of the libafl library code. When this Java class is initialised, the **static** initialisation block instructs the JVM to find and load the native, platform-dependant, copy of the libafl library.

```
public class AFL {
    /**
     * Load the native AFL library.
     */
    static {
        System.loadLibrary("afl");
    }

    /**
     * Initialise AFL library with an input and output directory as well as the
     * location of the target. Also executes a number of tests on the target (e.g.
     * whether the forkserver is and coverage bitmap work as expected) as well as some
     * calibration steps to determine the variation in execution time of the target
     * (used to detect hangs).
     *
     * @param inputDirectory  AFL input directory, must contain at least a
     *                        single valid test case used for
     *                        initialisation tests and calibration
     * @param outputDirectory AFL output directory, where AFL will save test
     *                        cases it deems "interesting"
     * @param target          location of the target binary, compiled with
     *                        AFL instrumentation code
     */
    public native void pre(String inputDirectory, String outputDirectory, String target);

    /**
     * Shut down AFL, including the forkserver.
     */
    public native void post();

    /**
     * Run the target with our given test case as input, and return the output.
     *
     * @param testcase the test case to be used as an input to the target
     *                 program
     * @return output produced by the target program
     */
    public native byte[] run(byte[] testcase);

    /**
     * Get the number of "interesting" test cases discovered in this run (called
     * queued_discovered by AFL internally).
     *
     * @return the number of test cases discovered in this run
     */
    public native int getQueuedDiscovered();

    /**
     * Get the coverage bitmap from the previous run, which is essentially a trace of
     * all edges (code branches) covered by the test case we just ran.
     *
     * @return the coverage (trace) bitmap for the previous run.
     */
    public native byte[] getTraceBitmap();
}
```

is in contrast to the default behaviour of LearnLib, which expects a *single-step SUL*: a target implementation that repeatedly accepts an input value and returns the associated output, and an explicit option to reset. Because this behaviour is not possible with AFL, we decided to simulate it by running the target program multiple times and comparing outputs between the previous and current run. However, this means that a trace with five steps will cause five target invocations, one for every step, which does not help performance. An improved method was implemented by also implementing our own Mealy membership oracle which can answer an entire membership query (input trace) at once, instead of in steps. This means that we can suffice with only a single run of the target process. However, our membership oracle will need to correlate every input symbol with its corresponding output symbol, which is not too complicated as long as the symbols in the output of the target have clear separators (e.g. spaces or newlines).

To implement equivalence testing using fuzzed test cases we implemented a new equivalence oracle, `AFLEQOracle`. This oracle loads test cases from an AFL output directory and parses them as a sequence of input symbols; any symbols that are not part of the alphabet are skipped. This can happen because fuzzing does not use a fixed input alphabet: for example, AFL find `-1` an interesting test case (because it triggers an error condition) while the input alphabet consists of only positive integers. These test cases are then evaluated against both the target program and the hypothesised model. When the output of the target program does not match the output predicted by the model, a counterexample is found and the learning process will continue refining the hypothesis.

3.2.2. Code coverage-based learning

We also implemented code coverage-based learning that uses the AFL edge (branch) coverage bitmap as the output of the target instead of the actual target output. This means that learning input/output behaviour is done purely this coverage bitmap, which means that we can also learn models for targets that do not have clear output behaviour. To make the bitmap human-readable, we represent each entry in the bitmap using its location in the bitmap using the code shown in Listing 3.2. In the next chapter we will evaluate this technique.

Listing 3.2: Edge coverage bitmap conversion

```
String output = friendlyBitmap(traceBitmap).stream().collect(Collectors.joining("#"));

public static List<String> friendlyBitmap(byte[] traceBitmap) {
    List<String> out = new ArrayList<>();
    for (int i = 0, traceBitmapLength = traceBitmap.length; i < traceBitmapLength; i++) {
        if (traceBitmap[i] != 0)
            out.add(String.format("%06d", i));
    }
    return out;
}
```

4

Results

In this chapter we first describe our work of replicating previous work on learning of TLS state machine. Secondly, we will test active learning on an obfuscated target program. Finally, will evaluate our approach for combining learning and fuzzing, as described in the previous chapter, on several benchmark problems.

4.1. Learning TLS state machines

To get acquinted with learning algorithms in general, we have tried to replicate a part of the research project by Ruiter and Poll [45] on learning state machine of TLS implementations.

We have used the original *TLSTestService* codebase of the research project to learn state machines for the OpenSSL 1.0.2g library code, for both the client and server components. To achieve this we had to fix several minor issues, which was a good way to gain insight into running learning experiments and the active learning process in general.

One of the issues was that a race condition existed in listener socket activation: the creation and binding of the server socket was done asynchronously (in another thread), which meant that the target (client) process would sometimes start when the socket was not yet bound. This in turn caused the learning process to hang, because the server code was now waiting for a connection attempt that was already made. After this problem was identified it was easily fixed by removing the separate thread, which ensures that the target program only starts after the socket is opened and bound.

Another issue was that LearnLib's resulting model was not very readable. This had two causes: one was the long TLS message names which were squashed together without any separator, easily fixed by adding an underscore character (_) after each message. The second was that the LearnLib code to create Graphviz DOT files created separate edges for every input/output transition. While this makes sense for small graphs, it turns larger graphs with many transitions into an unreadable spaghetti. We resolved this issue by creating a script that merges transitions together so that only a single edge is rendered from one state to another state.

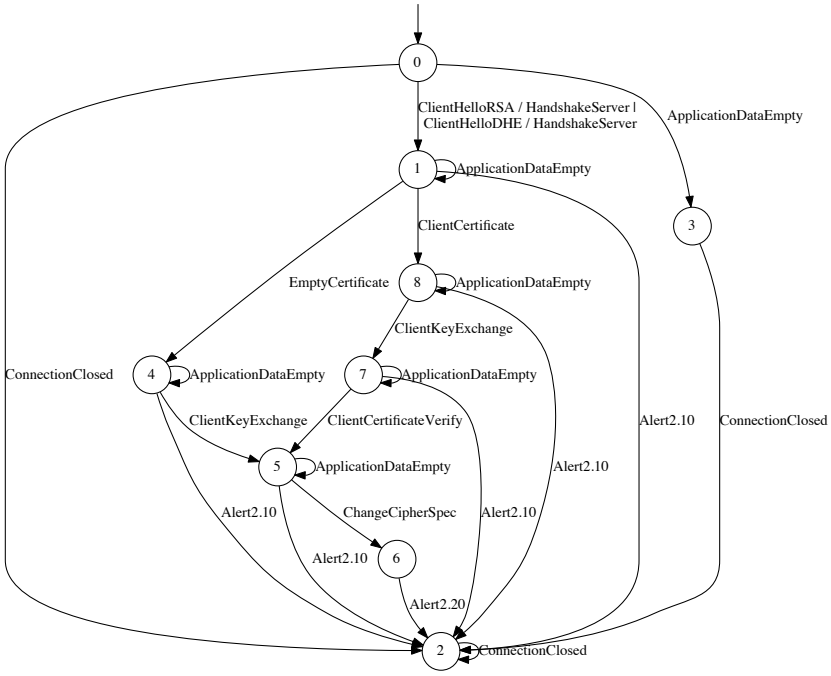

Figure 4.1: Learned model of the OpenSSL 1.0.2g server codebase using our version of the *TLSTestService* codebase. The model is very similar to models of other implementations as learned by Ruiter and Poll [45], and has simplified error handling compared to their model of OpenSSL 1.0.1j.

The resulting state machine when learning the state machine for the OpenSSL 1.0.2g server component is shown in Figure 4.1. From the learned model we can conclude that OpenSSL 1.0.2g has simplified error handling (fewer different transitions to error states) compared to the model of OpenSSL 1.0.1j by Ruiter and Poll. This would be considered a good thing, and brings OpenSSL closer to the clean state machine model of RSA BSAFE for Java from [45].

Our version of TLS state machine learning source code is available on GitHub[1].

[1] `https://github.com/praseodym/TLSTestService`

4.2. Simple State Machine

We created a simple state machine system written in C, with three states, an input alphabet (A, B, C) and an output alphabet (X, Y, Z). The (essentials) of this program are shown in Listing 4.2.

The source code of the program was then obfuscated using Tigress. Various obfuscation techniques are applied; the Tigress command used to transform the source code is given in Listing 4.1. Whereas the compiled binary of the original (unobfuscated) source code is clearly organised, as can be seen from the IDA Pro proximity view shown in Figure 4.3, the obfuscated binary is complex and very hard to analyse, as shown in Figure 4.4. A snippet of the obfuscated code can be found in Listing 4.3.

We then learned a model for this program, for now without any extensive fuzzing-based equivalence testing but using the W-method instead. The resulting graph of the modelled Mealy machine is shown in Figure 4.2. As expected, the learned model is equivalent to state machine model implemented in code (Listing 4.2).

Listing 4.1: Tigress command line that lists the obfuscation methods we have chosen: encoding of literals and arithmetic, opaque expressions, adding implicit control flows and control flow flattening.

```
tigress \
    --Transform=InitEntropy --Functions=main \
    --Transform=InitOpaque --Functions=main \
    --Transform=EncodeLiterals --Functions=* \
    --Transform=EncodeArithmetic --Functions=* \
    --Transform=AddOpaque --Functions=* \
    --Transform=AntiTaintAnalysis --Functions=* \
    --Transform=Flatten --Functions=* \
    --Transform=CleanUp \
    --out=${input}_obfuscated.c ${input}.c
```

Listing 4.2: Source code of a simple state machine program, without obfuscation applied. The `main` function (not listed) reads input and calls the `step` function (shown here), which implements a simple state machine. The output of this function is also the output of the program.

```
int current_state = 0;
char step(char input) {
  switch (current_state) {
    case 0:
      switch (input) {
        case 'A':
          current_state = 1;
          return 'X';
        case 'B':
          current_state = 2;
          return 'Y';
        case 'C':
          return 'Z';
        default:
          invalid_input();
      }
    case 1:
      switch (input) {
        case 'A':
          current_state = 3;
          return 'Z';
        case 'B':
          return 'Y';
        case 'C':
          return 'Z';
        default:
          invalid_input();
      }
    case 2:
      switch (input) {
        case 'A':
          return 'Z';
        case 'B':
          return 'Y';
        case 'C':
          current_state = 0;
          return 'Z';
        default:
          invalid_input();
      }
  }
  return 0;
}
```

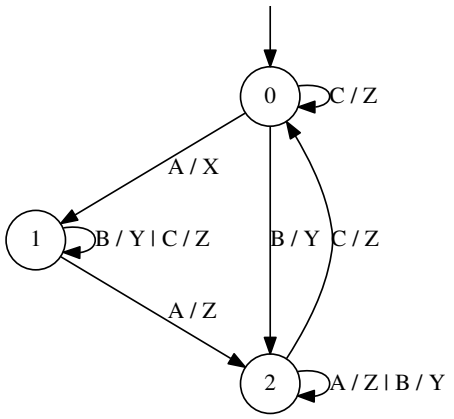

Figure 4.2: Learned Mealy machine model for the obfuscated version of the simple state machine (Listing 4.3). We can see that this models the behaviour of the original code (Listing 4.2) perfectly.

Listing 4.3: An obfuscated version of the simple state machine program listed in Listing 4.2. The code was obfuscated using the Tigress source-to-source obfuscator.

```
l___2314 = o___11 != o___20 ? 7 : 10;
while (1) {
  switch (l___2314) {
    case 12:
      o___28(2, o___16);
      l___2314 = 11 - ((o___11 != o___20) + (o___11 != o___20));
      break;
    case 15:
      l___2305 = scanf((char const */* __restrict  */) (o___19), &l___2303);
      l___2314 = 14 + !(o___11 == o___20);
      break;
    case 2:;
      l___2314 = (unsigned long) (o___20 != (struct t___8 *) 0UL)
          - (unsigned long) (o___11 == (struct t___8 *) 0UL);
      break;
    case 13:
      l___2306 = l___2307;
      l___2314 = 12 - ((o___20 == (struct t___8 *) 0UL)
          + (o___20 == (struct t___8 *) 0UL));
      break;
    case 1:
      o___13 = ((l___2304 & ~o___13) << 1) - (l___2304 ^ o___13);
      l___2314 = o___20 == (struct t___8 *) 0UL ? 8 : 8;
      break;
    case 3:
      l___2307 = o___12(l___2303);
      l___2314 = o___11 == (struct t___8 *) 0UL ? 13 & l___2304 : 13;
      break;
    case 7:;
      if ((unsigned int) ((((l___2304
          - (-1 & (o___20 != (struct t___8 *) 0UL))
              * (-1 | (o___20 != (struct t___8 *) 0UL)))
          + (-0x7FFFFFFF - 1)) + (((l___2304
          - (-1 & (o___20 != (struct t___8 *) 0UL))
              * (-1 | (o___20 != (struct t___8 *) 0UL)))
          + (-0x7FFFFFFF - 1)) >> 31)) ^ (((l___2304
          - (-1 & (o___20 != (struct t___8 *) 0UL))
              * (-1 | (o___20 != (struct t___8 *) 0UL)))
          + (-0x7FFFFFFF - 1)) >> 31)) >> 31U) {
        l___2314 = o___20 == (struct t___8 *) 0UL ? 7 : 6;
      } else {
        l___2314 = o___20 != (struct t___8 *) 0UL ? 5 : 7;
      }
      break;
    // ...
  }
}
```

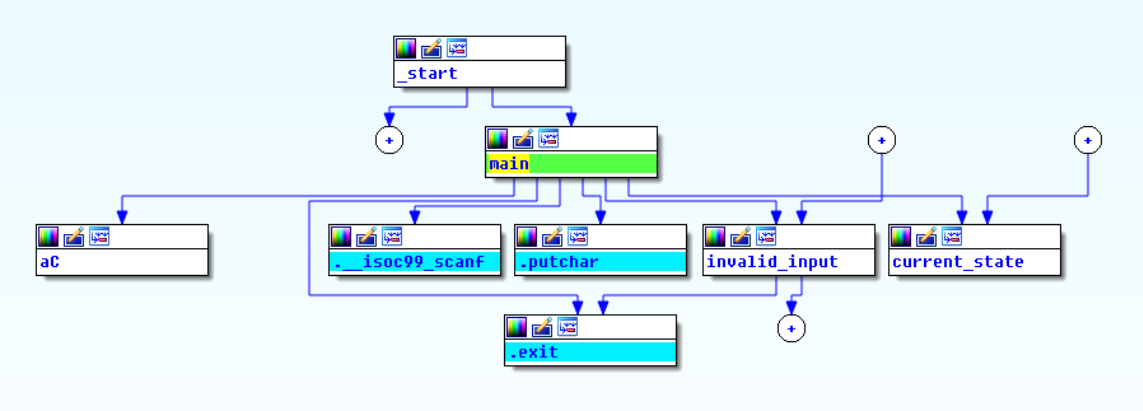

Figure 4.3: IDA Pro proximity view of the compiled simple state machine code (Listing 4.2); the main function is highlighted. No obfuscation was applied, and control flow structures are clearly shown.

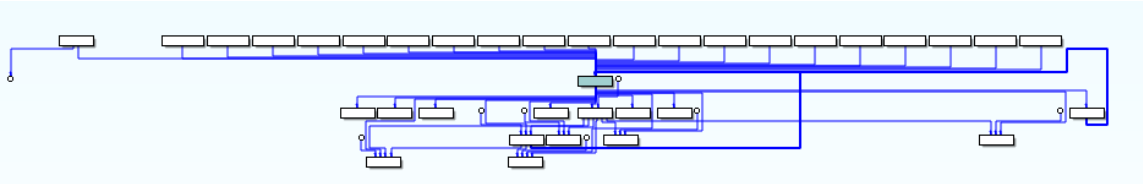

Figure 4.4: IDA Pro proximity view of the obfuscated and compiled simple state machine code (Listing 4.3); the main function is highlighted.

4.3. Forkserver performance

We then evaluated using the AFL forkserver (as described in subsection 2.5.4) to speed up target executions on the simple state machine that was described in the previous section. By running the learning experiment with the W-method for equivalence testing set to a high depth, we can vastly increase the amount of queries that are done to the target, which leads to more target executions. While this does not yield a better model for this target, it does allow us to evaluate the performance (learning time) of the algorithm.

We conducted two learning experiments, both with W-method equivalence with the maximum of 9. In the first experiment, we executed the target process directly from our Java code. In the second experiment, we used the AFL forkserver to execute the target. In both experiments, 590 556 queries were done to the target, and the exact same model was learned. However, where first experiment took 586 805 ms, the second experiment took 272 734 ms. Using the AFL forkserver thus leads to a 2.15× improvement in learning time for this target. For larger targets, we can expect the performance difference to be even higher, because larger programs take longer to initialise and this initialisation is skipped when using the AFL forkserver.

4.4. RERS2015 Challenge Problems

The Rigorous Examination of Reactive Systems (RERS) Challenge 2015 [18] is the 5th International Challenge on the Rigorous Examination of Reactive Systems and is co-located with the 15th International Conference on Runtime Verification. The event was be held in September 2015, in Vienna, Austria. RERS is designed to encourage software developers and researchers to apply and combine their tools and approaches in a free style manner to answer evaluation questions for specifically designed benchmark suites.

4.4.1. Description of the Challenge Problems

The Rigorous Examination of Reactive Systems (RERS) 2015 challenge consists of six obfuscated programs in two categories: three whitebox testing problems (problems 1 to 3) and three runtime verification problems (problems 4 to 6). These problems are published as Java and C source files; the contestant choose either of them depending on preference (or suitability with used tools). We will be using the C versions.

The problems in each category have increasing complexity: Problem 1 is a 36 kB source file, whereas Problem 3 is a 22 MB source file. The problems use simple set of integers (1 to 5 for Problem 1, 1 to 20 for Problem 3) as the input alphabet. The output is also an integer value.

Because these problem programs are essentially systems with simple deterministic input/output behaviour, they are well suited for modelling as a Mealy machine. However, note that this simple behaviour is not at all apparent from the source code or compiled binary code, because of the applied obfuscation techniques. An example of the source code is shown in Listing 4.4.

The actual challenge consists of verification of safety and liveness properties, in the form of 100 LTL (linear temporal logic) statements and 100 reachability statements. We will be focussing on the reachability statements; these are modelled with a label (e.g. `error_42`) that is printed to the

output (in addition to the integer output) when a certain state is reached. The challenge is to verify which of the 100 labels (states) can be reached and which ones cannot be reached.

Listing 4.4: An excerpt of the source code of the RERS2015 Challenge Problem 3. This excerpt shows one of the 2476 functions in the 22 MB source code. While we can see that this function prints '25' as an output, it is not clear how we can reach this function.

```
 void calculate_outputm1233(int input) {
    if((((a1158!=1) && (((23 == a2294[4]) && ((a1991!=1) && ((47 == a2222[3])
       && (((cf==1) && 346 < a1523) && 249 < a147)))) && (21 == a1788[4])))
       && ((a876 == a932[5]) && (input == inputs[19] && ((a1852!=1)
       && ((a20 75 == 32) && (a2013 == a2196[1])))))))) {
        a70 += (a70 + 20) > a70 ? 2 : 0;
        cf = 0;
        if((((63 < a1958) && (263 >= a1958)) && ((!(52 == a554[0])
           && ((a1031 == 7) && (61 == a2214[1])))) || (a1756 == a1862[1])))) {
        a1950 = 1;
        a147 = (((((a147 * a1523) % 14999) - -9103) * 1) - 38794);
        a56 = a158[0];
        a458 = 36 ;
        a98 = a26[0];
        a2222 = a2057;
        a1852 = 1;
        a2124 = 33 ;
        a388 = a344[1];
        }else {
        a1852 = 1;
        a1950 = 1;
        a1991 = 1;
        a2039 = a2049[(a1821 / a1833)];
        a147 = ((((((a147 * a1770) % 14999) + -25750) * 1) * 10) / 9);
        a1300 = 1;
        a1394 = 10;
        a1813 = ((a1936 * a1682) + -75);
        a56 = a158[(a1394 - 7)];
        a737 = a648;
        }printf("%d\n", 25); fflush(stdout);
    }
}
```

4.4.2. Results with Active Learning

To evaluate the performance of active learning, we learned the Mealy machine model of the RERS2015 challenge problems with both L* and TTT learning algorithms, and both with W-method equivalence testing and our implementation of using the AFL fuzzer for equivalence testing. We can then compare the number of states learned by both methods: because the learned state machine model is always minimal (a basic premise of the learning algorithm), having more states in the model means more behaviour of the target program is modelled. In addition, we can check how many RERS challenge reachability labels were found, and because the solutions to the problems are available can also check how many of of the total possible reachability labels we found. Furthermore, we can also compare learning performance in terms of learning time and the number of queries needed (lower is better) and compare this relative to the quality of the model.

An overview of our results is shown in Table 4.1. For several learned models we have also rendered pictures that visually show the difference between the models learned with W-method equivalence versus the models learned using our fuzzing approach. In all cases, using fuzzing equivalence delivers better quality models (more states, more reachability labels found) in a shorter learning time. For some problems the results are significant: for Problem 4, we learned about 343× more state using fuzzing than using the W-method (21 versus 7402 states) and found 21 reachability labels using fuzzing versus only 1 using the W-method. For other problems, the difference is less striking: for Problem 3, we learned only about 1.4× more states (798 versus 1094 states) and 3 more reachability labels using fuzzing.

For Problems 4, 5 and 6 we used only the TTT learning algorithm in combination with our fuzzing equivalence test, because the L* method did not finish within a reasonable time (several days). The maximum W-method depth we chose was also limited by the execution time; using a larger maximum depth leads to an exponentially longer equivalence testing time. For depth values with reasonable runtime (shorter than a day), increasing the depth did not lead to learning of a significant number of new states.

In the rendered figures, we excluded the sink state that is reached when an invalid transition is made by giving an invalid input. In this sink state, the program will only return `invalid_state` as an output. Because many other states in the program have a transition to this state for some input value, visualising the models with this state creates a cluttered picture.

One remark is that the learning time we report only includes the time the learning process ran, not including the time our fuzzer ran. We ran the AFL fuzzer on each problem for 24 to 48 hours, and the test cases that were generated during that time were used for equivalence testing using the learning process. If we include the fuzzing time in our comparison, the W-method is still outperformed because of its extremely long running times when trying to achieve similar coverage.

Table 4.1: Results for active learning applied to the RERS2015 challenge problems. The second column shows the learning algorithm chosen (L* or TTT) and the method for equivalence testing (either the W-method or our implementation of equivalence using the AFL fuzzer). The third column shows the total number of states in the state machine that was learned and the fourth column lists the number of reachability labels that we reached (out of the total possible number of labels). The fifth column lists the time needed to learn the model (hh:mm:ss). Finally, the sixth column lists the total number of queries needed (during learning and equivalence testing).

Target	Method	States	Reachability	Time	Queries	Figure
Problem 1	TTT, W-method 1	25	19/29	00:00:04	7342	-
Problem 1	L*, W-method 8	25	19/29	13:49:10	246 093 350	Figure 4.5
Problem 1	TTT, fuzzing eq.	334	29/29	00:00:21	16 731	-
Problem 1	L*, fuzzing eq.	1027	29/29	00:44:40	2 860 990	Figure 4.6
Problem 2	TTT, W-method 1	188	15/30	01:00:05	8 156 730	-
Problem 2	L*, W-method 3	195	15/30	17:05:59	239 851 191	Figure 4.7
Problem 2	TTT, fuzzing eq.	2985	24/30	00:13:06	412 340	-
Problem 2	L*, fuzzing eq.	3281	24/30	13:28:42	42 120 554	Figure 4.8
Problem 3	L*, W-method 1	798	16/32	14:15:36	1 421 330 223	Figure 4.9
Problem 3	TTT, fuzzing eq.	1054	19/32	00:13:27	698 409	-
Problem 3	L*, fuzzing eq.	1094	19/32	13:28:42	23 464 145	Figure 4.10
Problem 4	TTT, W-method 7	21	1/23	04:11:13	51 760 913	Figure 4.11
Problem 4	L*, W-method 7	21	1/23	03:10:51	51 759 741	-
Problem 4	TTT, fuzzing eq.	7402	21/23	00:16:48	458 763	Figure 4.12
Problem 5	L*, W-method 1	183	15/30	00:13:17	2 203 813	-
Problem 5	TTT, fuzzing eq.	3376	24/30	00:08:00	416 943	-
Problem 6	L*, W-method 1	671	16/32	21:33:10	889 677 067	-
Problem 6	TTT, fuzzing eq.	3909	23/32	00:45:00	1 804 595	-

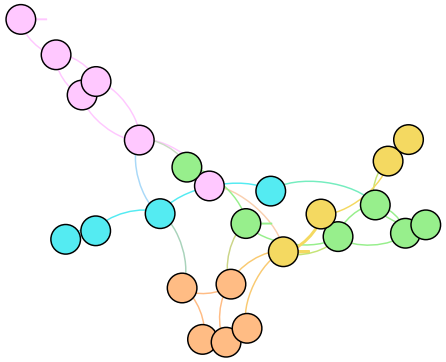

Figure 4.5: Learned Mealy machine model of RERS2015 Problem 1 using L* and W-method equivalence (depth 8). The learned state machine model has 25 states.

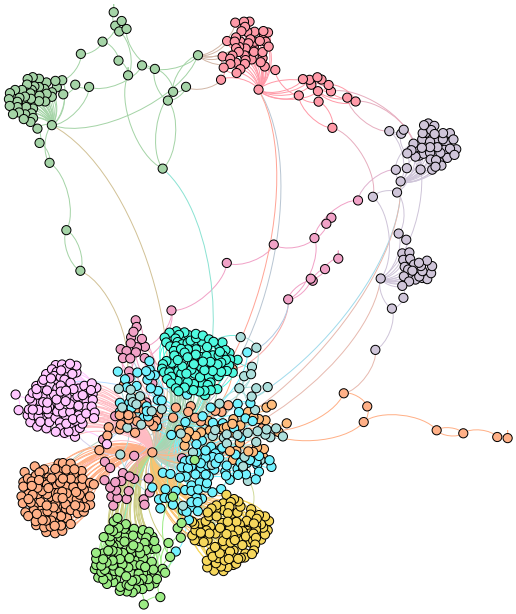

Figure 4.6: Learned Mealy machine model of RERS2015 Problem 1 using L* and fuzzing equivalence. The learned state machine model has 1027 states, compared to 25 when learned using the W-method.

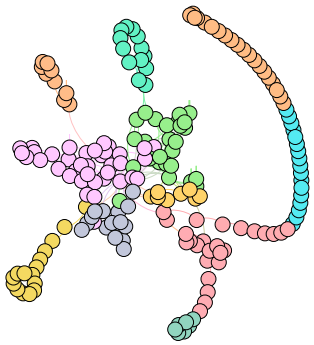

Figure 4.7: Learned Mealy machine model of RERS2015 Problem 2 using L* and W-method equivalence (depth 3). The learned state machine model has 195 states.

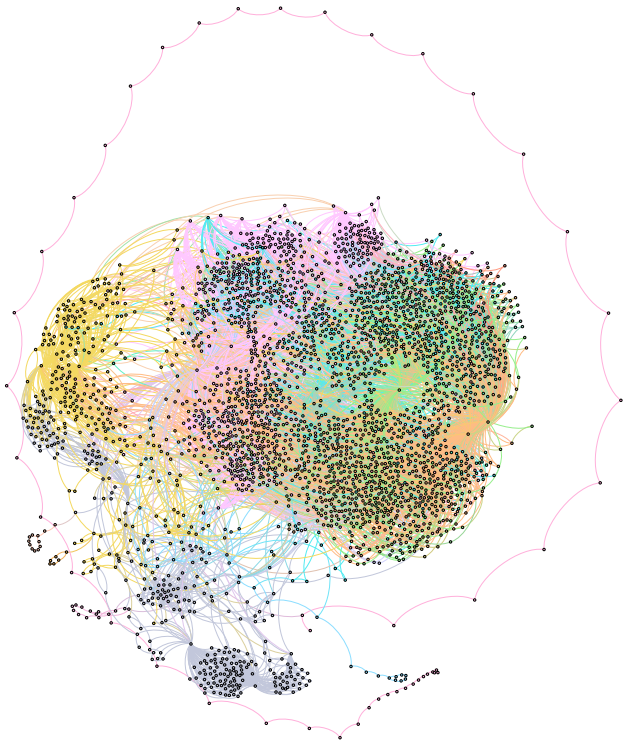

Figure 4.8: Learned Mealy machine model of RERS2015 Problem 2 using L* and fuzzing equivalence. The learned state machine model has 3281 states, compared to 195 when learned using the W-method.

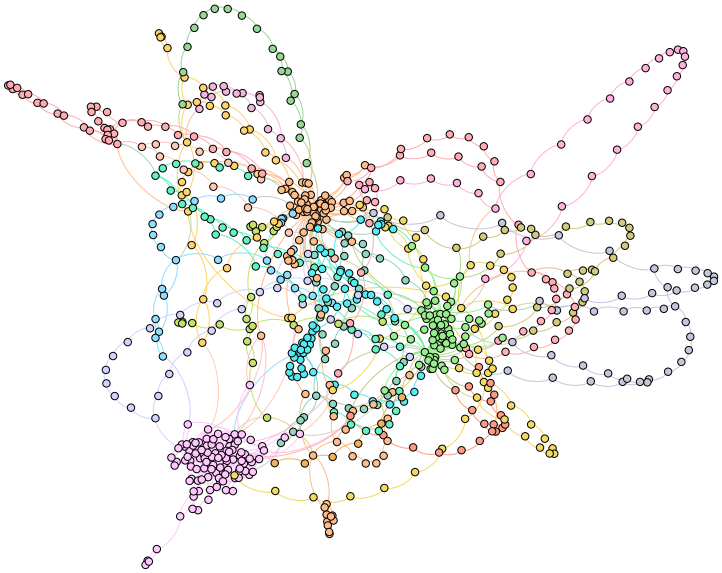

Figure 4.9: Learned Mealy machine model of RERS2015 Problem 3 using L* and W-method equivalence (depth 1). The learned state machine model has 798 states.

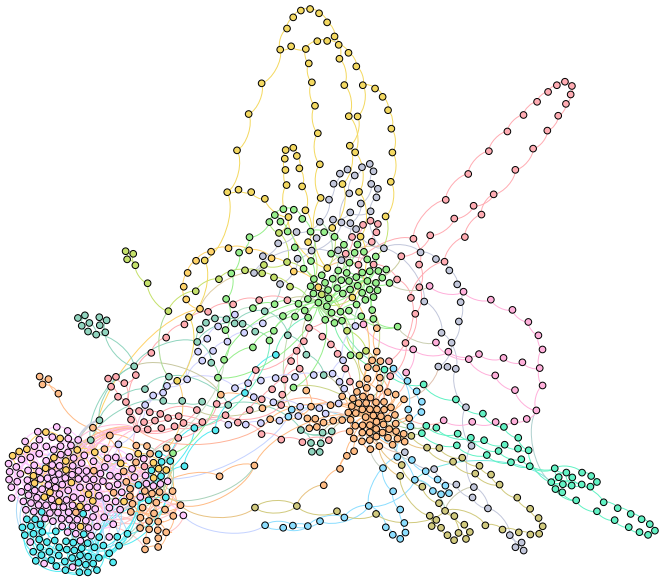

Figure 4.10: Learned Mealy machine model of RERS2015 Problem 3 using L* and fuzzing equivalence. The learned state machine model has 1094 states, compared to 798 when learned using the W-method.

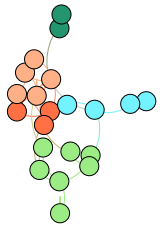

Figure 4.11: Learned Mealy machine model of RERS2015 Problem 4 using TTT and W-method equivalence (depth 7). The learned state machine model has 21 states.

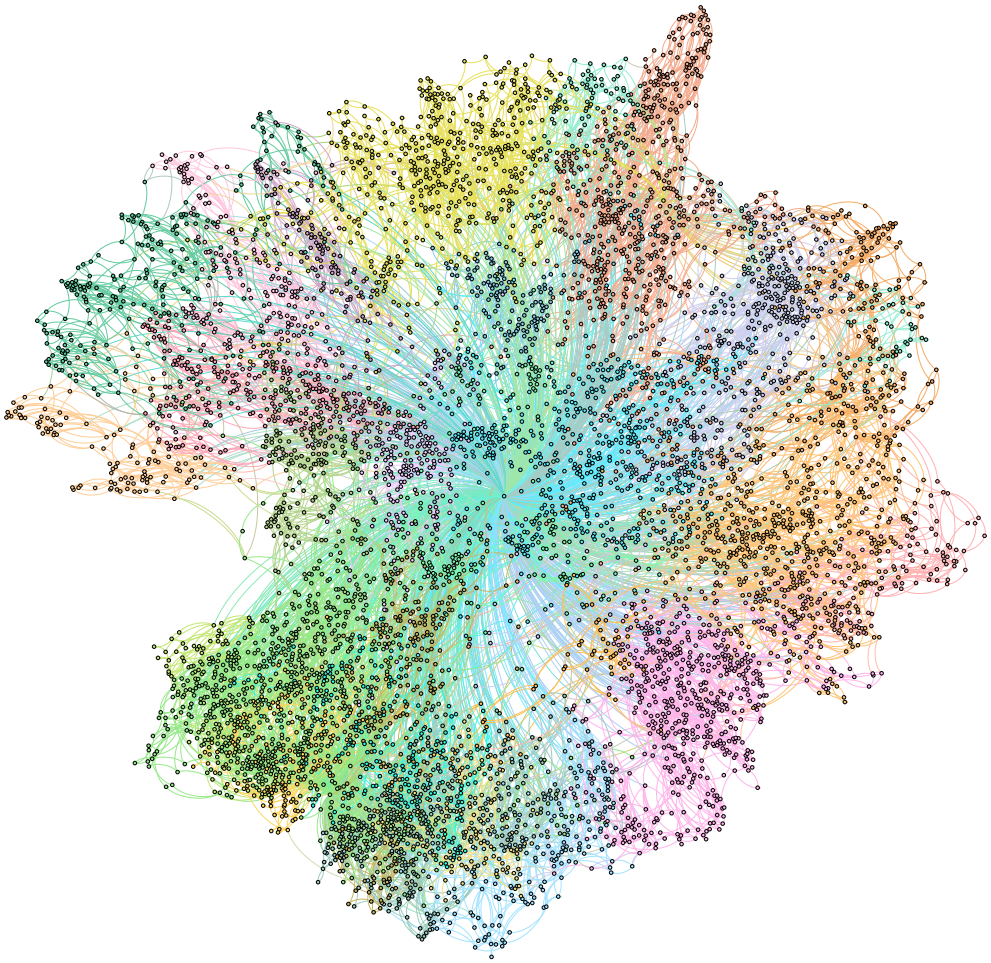

Figure 4.12: Learned Mealy machine model of RERS2015 Problem 4 with TTT and fuzzing equivalence. The learned state machine model has 7402 states, compared to 21 when learned using the W-method.

4.5. Coverage-Based Learning

To validate that AFL's control flow edge coverage (as described in subsection 2.5.2) works as expected, we tested it on the simple state machine implementation (see Figure 4.2 and Listing 4.2).

As can be seen in Listing 4.5, no additional edges are returned for traces that do not visit other states compared to the previous trace. The edge coverage thus works as expected.

Unfortunately, using active learning with the edge coverage bitmap as output leads to a state explosion. Learning the model for our simple state machine yields 914 states, where only three are expected. However, we expect that this behaviour can be improved by using a clustering algorithm to combine related states together. We think that this is something that would be an interesting topic for future research.

Listing 4.5: Traces from control flow edge coverage on test cases for the simple state machine program from section 4.2. The test cases that do not trigger new behaviour, i.e. the program stays in the same state, do not introduce new edges into the trace bitmap.

```
Test case: [A C A] -> [X Z Z]
000745#011327#019073#021339#023122#027961#032615#032664#034435#041136#
044245#046173#046548#046782#053335#059227
New trace edges: 2

Test case: [A C A B] -> [X Z Z Y]
000745#011327#019073#021178#021339#023122#027961#029693#032615#032664#
034435#041136#044245#046173#046548#046782#053335#059227#063757
New trace edges: 3

Test case: [A C A B B] -> [X Z Z Y Y]
000745#011327#019073#021178#021339#023122#027961#029693#032615#032664#
034435#041136#044245#046173#046548#046782#053335#059227#063757
New trace edges: 0

Test case: [A C A B B B] -> [X Z Z Y Y Y]
000745#011327#019073#021178#021339#023122#027961#029693#032615#032664#
034435#041136#044245#046173#046548#046782#053335#059227#063757
New trace edges: 0
```

5

Conclusions and Future Work

In this chapter we summarise the main findings of our work, describe threats the validity of our work and provide some directions for future research.

5.1. Conclusions

RQ 1 *Can we use learning algorithms to gain insight into the behaviour of obfuscated programs?*

We have created a program that implements a state machine and which was then obfuscated using the Tigress obfuscator. With more traditional reverse engineering methods, like using IDA Pro to disassemble the program, it is not trivial to gain insight into the behaviour of this program. Through the use of our active learning prototype tool, we were able to learn the entire state machine as implemented by the program. We were also able to learn models for all of the RERS2015 challenge problems, which are heavily obfuscated programs that implement behaviour that can be modelled as a state machine. We can thus conclude that it is possible to gain insight into the behaviour of several obfuscated programs by using learning algorithms.

RQ 2 *Can we combine learning algorithms with fuzzing techniques?*

By using the AFL fuzzer as a way to generate test cases for equivalence testing we have combined learning algorithms with fuzzing techniques. In our active learning tool prototype, the test cases generated by the fuzzer are used during the equivalence testing phase, where an hypothesis state machine created by the learning algorithm is checked against test cases to find counterexamples that have a difference in output between the real program and the hypothesised state machine. A counterexample is used by the learning algorithm to create a new, refined hypothesis. This process is repeated until no more counterexamples are found and the learned model incorporates all information from our set of fuzzed test cases.

RQ 3 *Is it possible to use insight in the execution of a program to improve the quality of the learned model?*

The AFL fuzzer generates test cases by means of mutation and using a genetic algorithm to rank test cases. Test cases that cover new edges of the control flow in the target program will be retained and

used in further mutation cycles so as to create more test cases. This knowledge of what edges are covered is gathered through compile-time or runtime instrumentation of the target program, which means that we use insight in the execution of a program to create find test cases. Because these test cases are used by our active learning tool to improve the hypothesis of the learning algorithm through finding of counterexamples in the equivalence testing phase, this improves the quality of the learned model.

We have evaluated our approach on the six RERS2015 challenge problems. For these problems we found between 1.4× and 343× more states when learning a model with fuzzing compared to learning where the W-method is used for equivalence testing. In addition, because the test cases are generated by the fuzzer before the learning process is started, we need fewer queries to the target program to find counterexamples during the actual learning process. This leads to a shorter learning time.

5.2. Threats to Validity

In this section, we discuss the limitations and threats that affect the validity of our research. For these threats, we show how we mitigated them or propose future work to address them.

Threats to construct validity mainly concern errors in the correctness of learning algorithms and their output, the learned models. Our prototype tool relies on the LearnLib library, which implements a number of learning algorithms. In our evaluation, we used the implementations of the L* and TTT active learning algorithms. When these implementations were to contain bugs, the models learned by our tool could be affected. However, because LearnLib was extensively used in other research, we think it is highly unlikely that such bugs exist. In addition, we verified that all our learned models were minimal finite state machines, i.e. they do not contain any duplicate states.

Threats to internal validity relate to how we performed our evaluation. We used the AFL fuzzer to generate test cases for our target problems, which is done through a series of deterministic and random mutation stages. Because there is randomness involved, it could be that the fuzzer finds interesting test cases that it would not have found when ran another time. To check if this was happening for our evaluation, we ran every fuzzing task at least twice and compared the test sets. For the RERS2015 challenges, this was done by comparing reachability labels found by the test sets, where we found no significant differences between fuzzer runs.

Threats to external validity concern the generalisability of our results. Our conclusions are mainly based on the results of benchmarking with the RERS2015 challenge problems, where we see significant improvement over other evaluated methods. For other target programs, our approach might not work as well. Furthermore, we only compared against the W-method. Further study is needed to evaluate how well our approach works on other targets and how it compares to other testing algorithms, such as the adaptive distinguishing sequences method by Lee and Yannakakis [34].

5.3. Future Work

There are several directions in which this work can be extended in future research.

Passive learning It is possible to use passive learning techniques on test cases generated through fuzzing. It would be interesting to compare this to active learning; notable points for comparison could be learning time and the quality of the learning model in terms of overall correctness.

The overall approach would be to generate test cases using fuzzing. For these test cases, the target program is ran to determine its output so that a trace of input and output is created. A collection of these traces can be used by a passive learning tool (such as DFASAT) to learn a state machine model.

Learning directly from coverage In section 4.5 we have seen that it is possible to create traces from the AFL coverage bitmap. By using e.g. a clustering algorithm on inputs and their associated coverage traces, we think that it would be possible to learn a state machine model of a program purely from its code coverage.

Visualising code coverage in learned models The idea is give insight into a (learned) state machine in relation to the code that is executed. For example, given a certain learned state machine model, one could provide code insight by using coverage or trace data to show what code is executed for a given input sequence. This could be combined with tainting to show where exactly input values are being used to influence behaviour.

AFL on LLVM IR The AFL-LLVM compiler could be adapted to work on LLVM IR [37] (generally encoded as LLVM Bitcode [36], which would allow it to instrument existing binaries if they can be disassembled to LLVM IR. This would give large performance benefits over AFL's QEMU fuzzing, the current go-to solution for fuzzing binary-only programs that cannot be recompiled with instrumentation.

Bibliography

[1] Fides Aarts, Joeri De Ruiter, and Erik Poll. 2013. Formal Models of Bank Cards for Free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 461–468. http://dx.doi.org/10.1109/ICSTW.2013.60

[2] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (nov 1987), 87–106. http://dx.doi.org/10.1016/0890-5401(87)90052-6

[3] Arxan Technologies Inc. 2015. Arxan Application Protection. (2015). https://www.arxan.com/products/application-protection/

[4] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W Smith. 2013. The Page-Fault Weird Machine: Lessons in Instruction-less Computation. In *Proceedings 7th USENIX Workshop on Offensive Technologies*. USENIX, Washington, D.C. https://www.usenix.org/conference/woot13/workshop-program/presentation/Bangert

[5] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2001. On the (im)possibility of obfuscating programs. In *Advances in cryptology—CRYPTO 2001*. Springer, 1–18.

[6] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-lavaud, Cedric Fournet, and Markulf Kohlweiss. 2015. A Messy State of the Union: Taming the Composite State Machines of TLS. *IEEE Symposium on Security and Privacy* (2015), 1–19. https://www.ieee-security.org/TC/SP2015/papers/6949a535.pdf

[7] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R Piegdon. 2010. libalf: The Automata Learning Framework. *Computer Aided Verification* 6174, Procope (2010), 360–364.

[8] Joppe W Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. 2015. Differential Computation Analysis: Hiding your White-Box Designs is Not Enough. (2015), 1–22.

[9] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices* 10, 6 (jun 1975), 234–245. http://dx.doi.org/10.1145/390016.808445

[10] Chih-Fan Chen, Theofilos Petsios, Marios Pomonis, and Adrian Tang. 2013. CONFUSE: LLVM-based Code Obfuscation. (2013).

[11] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C Van Oorschot. 2003a. A white-box DES implementation for DRM applications. In *Digital Rights Management*. Springer, 1–15. http://dx.doi.org/10.1007/b11725

[12] Stanley Chow, Philip a Eisen, Harold Johnson, Paul C Van Oorschot, and Paul C Van Oorschot. 2003b. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*. Springer, 250–270. http://dx.doi.org/10.1007/3-540-36492-7_17

[13] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A Zakharov. 2001. An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. In *Information Security*. Springer, 144–155. http://dx.doi.org/10.1007/3-540-45439-X_10

[14] T.S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering* SE-4, 3 (may 1978), 178–187. http://dx.doi.org/10.1109/TSE.1978.231496

[15] Christian Collberg, Sam Martin, Jonathan Myers, and Bill Zimmerman. 2014. The Tigress diversifying C virtualizer. (2014). http://tigress.cs.arizona.edu/

[16] Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education.

[17] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations*. Technical Report 148. Department of Computer Science, The University of Auckland, New Zealand. 36 pages. http://dx.doi.org/10.1.1.38.9852

[18] RERS 2015 Committee. 2015. The Rigorous Examination of Reactive Systems (RERS) Challenge 2015. (2015). http://www.rers-challenge.org/2015/

[19] Bruce Dang, Bachaalany Elias Gazet Alexandre, and Sébastien Josse. 2014. *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. John Wiley & Sons.

[20] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 4963 LNCS. Springer, 337–340. http://dx.doi.org/10.1007/978-3-540-78800-3_24

[21] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*. ACM Press, New York, New York, USA, 51. http://dx.doi.org/10.1145/1455770.1455779

[22] Stephen Dolan. 2013. mov is Turing-complete. (2013), 1–4. http://www.cl.cam.ac.uk/

[23] Chris Domas. 2015. M/o/Vfuscator2. (2015). https://github.com/xoreaxeaxeax/movfuscator

[24] Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. 2007. Cryptanalysis of white box DES implementations. In *Selected Areas in Cryptography*. Springer, 278–295.

[25] Felix Gröbert, Carsten Willems, and Thorsten Holz. 2011. Automated Identification of Cryptographic Primitives in Binary Programs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 6961 LNCS. 41–60. http://dx.doi.org/10.1007/978-3-642-23644-0_3

[26] Yoann Guillot and Alexandre Gazet. 2010. Automatic binary deobfuscation. *Journal in Computer Virology* 6, 3 (aug 2010), 261–276. http://dx.doi.org/10.1007/s11416-009-0126-4

[27] Marijn J H Heule and Sicco Verwer. 2010. Exact DFA identification using SAT solvers. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6339 LNAI (2010), 66–79. http://dx.doi.org/10.1007/978-3-642-15488-1_7

[28] Erik Hjelmvik and Wolfgang John. 2010. *Breaking and Improving Protocol Obfuscation*. Technical Report. Chalmers University of Technology. 31 pages.

[29] Susan Horwitz. 1997. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems* 19, 1 (jan 1997), 1–6. http://dx.doi.org/10.1145/239912.239913

[30] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. 307–322. http://dx.doi.org/10.1007/978-3-319-11164-3_26

[31] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, Brecht Wyseur (Ed.). IEEE, 3–9. http://dx.doi.org/10.1109/SPRO.2015.10

[32] Michael J Kearns and Umesh Virkumar Vazirani. 1994. *An introduction to computational learning theory*. MIT press.

[33] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 75–86. http://dx.doi.org/10.1109/CGO.2004.1281665

[34] David Lee and Mihalis Yannakakis. 1994. Testing Finite-State Machines: State Identification and Verification. *IEEE Trans. Comput.* 43, 3 (1994), 306–320. http://dx.doi.org/10.1109/12.272431

[35] E.I. Lin, A.M. Eskicioglu, R.L. Lagendijk, and E.J. Delp. 2005. Advances in Digital Video Content Protection. *Proc. IEEE* 93, 1 (jan 2005), 171–183. http://dx.doi.org/10.1109/JPROC.2004.839623

[36] LLVM Project. 2016a. LLVM Bitcode File Format. (2016). http://llvm.org/docs/BitCodeFormat.html

[37] LLVM Project. 2016b. LLVM Language Reference Manual. (2016). http://llvm.org/docs/LangRef.html

[38] George H. Mealy. 1955. A method for synthesizing sequential circuits. *The Bell System Technical Journal* 34, 5 (sep 1955), 1045–1079. http://dx.doi.org/10.1002/j.1538-7305.1955.tb03788.x

[39] Morpher. 2015. Morpher Obfuscation Service. (2015). http://morpher.com/

[40] Peter Oehlert. 2005. Violating assumptions with fuzzing. *Security & Privacy, IEEE* 3, 2 (2005), 58–62.

[41] J. Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang. 2000. Experience with software watermarking. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. IEEE Comput. Soc, 308–316. http://dx.doi.org/10.1109/ACSAC.2000.898885

[42] Inc. Pivotal Software. 2016. Spring Framework. (2016). https://projects.spring.io/spring-framework/

[43] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. 2009. LearnLib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer* 11, 5 (nov 2009), 393–407. http://dx.doi.org/10.1007/s10009-009-0111-8

[44] Rolf Rolles. 2009. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies (WOOT)*. 1. http://dl.acm.org/citation.cfm?id=1855876.1855877

[45] J De Ruiter and E Poll. 2015. Protocol state fuzzing of TLS implementations. In *USENIX Security*. 193–206. http://www.cs.bham.ac.uk/

[46] Hex-Rays SA. 2015. Hex-Rays decompiler. (2015). https://www.hex-rays.com/products/decompiler/

[47] Eloi Sanfelix, Cristofaro Mune, and Job De Haas. 2015. Unboxing the White-Box. (2015).

[48] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. IEEE, 317–331. http://dx.doi.org/10.1109/SP.2010.26

[49] Federico Scrinzi. 2015. Behavioral Analysis of Obfuscated Code. (2015). http://essay.utwente.nl/67522/

[50] Koushik Sen. 2007. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*. ACM, ACM Press, New York, New York, USA, 571. http://dx.doi.org/10.1145/1321631.1321746

[51] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13*. ACM Press, New York, New York, USA, 263. http://dx.doi.org/10.1145/1081706.1081750

[52] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, Reston, VA. http://dx.doi.org/10.14722/ndss.2015.23294

[53] Axel Souchet. 2013. Obfuscation of steel: meet my Kryptonite. 2013 (2013), 1–24. http://download.tuxfamily.org/overclokblog/Obfuscationofsteel

[54] M. P. Vasilevskii. 1973. Failure diagnosis of automata. *Cybernetics* 9, 4 (1973), 653–665. http://dx.doi.org/10.1007/BF01068590

[55] VMProtect Software. 2015. VMProtect. (2015). http://vmpsoft.com/

[56] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. *Software tamper resistance: Obstructing static analysis of programs*. Technical Report. Technical Report CS-2000-12, University of Virginia, 12 2000. 1–18 pages. http://dx.doi.org/10.1.1.35.2337

[57] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (jul 1984), 352–357. http://dx.doi.org/10.1109/TSE.1984.5010248

[58] WhiteCryption. 2015. whiteCryption Code Protection. (2015). http://www.whitecryption.com/code-protection/

[59] Brecht Wyseur. 2012. White-box cryptography: hiding keys in software. *NAGRA Kudelski Group* (2012).

[60] Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. 2007. Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings. In *Selected Areas in Cryptography*. Springer Berlin Heidelberg, Berlin, Heidelberg, 264–277. http://dx.doi.org/10.1007/978-3-540-77360-3_17

[61] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. IEEE, 297–300. http://dx.doi.org/10.1109/BWCCA.2010.85

[62] Michał Zalewski. 2014a. Binary fuzzing strategies: what works, what doesn't. (2014).

[63] Michał Zalewski. 2014b. Fuzzing random programs without execve(). (2014). https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html

[64] Michał Zalewski. 2014c. Pulling JPEGs out of thin air. (2014). https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html

[65] Michał Zalewski. 2015. American Fuzzy Lop (AFL) fuzzer. (2015). http://lcamtuf.coredump.cx/afl/

[66] William Zhu, Clark Thomborson, and Fei-Yue Wang. 2006. Applications of homomorphic functions to software obfuscation. In *Intelligence and Security Informatics*. Springer, 152–153.