Fuzzing and Information Flow

Daniel Jordan Blackwell

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of
University College London.

January 15, 2025

I, Daniel Jordan Blackwell confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Maintaining the secrecy of confidential information in software systems is a common problem; from file ownership and read-write permissions in filesystems to user account specifics in web applications. The academic research area of information flow control has worked on solving the problem of validating that software satisfies security policies for decades. Much of this work has required either significant developer intervention, failed to scale to real-world software systems, or is not automatable.

This thesis applies the automated testing technique fuzzing to the problem of information flow control; in particular looking for leaks of confidential information through program outputs, as opposed to through side-channels. As a system-level testing approach, the level of developer intervention is relatively low and scalability high. The contributions of the thesis are divided into three distinct chapters:

Firstly, how a fuzzer can be used to detect instances of confidential information being leaked. The basis of this approach is hypertesting; in which finding two inputs that differ only in their secret parts, but result in differing

program outputs indicates a leak. The produced tool, LeakFuzzer, is evaluated on a newly collected set of benchmarks including 9 real-world programs containing CVEs classified as information leaks. These are up to 905,000LoC in size, and thus test scalability.

The next chapter extends this to include estimates of the quantity of information leaked through program outputs. Again the produced tool, NIFuzz, is evaluated on range of programs including CVEs.

The final chapter looks at improving the efficiency of grey-box fuzzers. The developed approach makes use of knowledge of the target program's control flow graph in order to better decide how to divide up the mutation effort. The technique and its implementation, PrescientFuzz, are applicable not only to information flow control; and PrescientFuzz outperforms other fuzzers at achieving program coverage on the Fuzzbench benchmark suite.

Impact Statement

One of the focuses of this thesis, information flow control, had early adoption in national defence where the confidential, secret and top-secret classifications are used. Since then, the spread of multi-user computer systems has ensured that almost everyone has had some exposure to the concept through file systems and online accounts. Unfortunately, the automated validation of such policies is still far from widespread, and vulnerabilities that result in confidential information leaks are still regularly reported. The scalable, automated approaches described in this thesis can be directly applied to searching for these vulnerabilities. There are 2 tools for detecting information flow control errors that have been made publicly available on GitHub as part of this thesis [1][2].

In industry, the tools could be used as part of an internal validation pipeline by software maintainers in order to test their own software. Externally, they could prove useful to penetration testers, whose job it is to simulate the actions of a cyberattack; or to security researchers, who aim to discover and

[1]LeakFuzzer: `https://github.com/DanBlackwell/LeakFuzzer`
[2]NIFuzz: `https://github.com/DanBlackwell/NIFuzz`

responsibly disclose software security flaws.

In academia, this work furthers the efforts of the information flow control community and is a large step towards a plug-and-play approach to detecting these vulnerabilities. Work could be done to automate these even further, with Google's 'oss-fuzz-gen' project to automatically generate fuzzing harnesses using LLMs being a potential source of inspiration. The Secure Information Flow Faults benchmark suite should prove useful in the evaluation of future information leakage detection techniques.

The final chapter of the thesis contributes a more general approach to improving grey-box fuzzer efficiency; again the implementation is publicly available [3]. This work could be integrated into the LibAFL fuzzing library upon which it is based; this would make the benefits available immediately to all users – a group made up of industrial users, academics and hobbyists.

As a software engineering thesis, this work is not bound by borders and is internationally applicable.

[3]PrescientFuzz: `https://github.com/DanBlackwell/PrescientFuzz`

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisors, David Clark and Ingolf Becker, for giving me a different perspective on things; and whose expertise, insight, and patience have guided me throughout this process.

I would also like to extend my thanks to the CDT Cybersecurity Directors, David Pym, Madeleine Carr, and Shane Johnson, for their leadership and for securing the funding that enabled me to pursue this research.

Also Justyna Petke, Myra B. Cohen and Ibrahim Mesecan for allowing me to get stuck in and start collaborating so early in my PhD, I have no doubt that this sped up my development as a researcher greatly.

I'd like to thank Bill Langdon for being there with advice, support and interesting tales.

On a personal note, I am immensely grateful to Chloe and Loki for their unwavering support and for putting up with me during the toughest moments.

Lastly, to my parents: I really appreciate your sacrifices that made it possible for me to be here where I am today.

Contents

1 Introduction **1**

1.1 Thesis Structure 5

2 Background **6**

2.1 High-Level Introduction 6

2.1.1 Information Flow Control 6

2.1.2 Fuzzing 11

2.2 Thesis Motivation 14

2.3 Grey-Box Fuzzing In-Depth 16

2.3.1 What really is an input? 16

2.3.2 Fuzzing Harnesses 17

2.3.3 Architecture of an AFL-based Fuzzer 18

2.3.4 Decision Engine 27

3 Related Work **29**

3.1 Information Flow Control Background 29

3.2 Formal Verification Methods for Information Flow Control . . 33

3.2.1 Secure Flow Typing (Language Based Security) 33

3.2.2 Symbolic Execution & Model Checking 35

3.2.3 Taint Analysis 37

3.3 Dynamic Methods for Information Flow Control 38

3.3.1 Assertion based Information Flow Policy Enforcement 38

3.3.2 Hypertesting 39

3.3.3 Self-Composition 41

3.3.4 Statistical Approaches 43

3.4 Side-Channel Leakage 44

3.5 Early Applications of Fuzzing to Side-Channel Leakage 47

3.6 Fuzzing 51

3.6.1 Sanitizers 55

3.7 Fuzzing for Information Leaks 56

3.8 Summary 56

4 Detecting Instance of Confidential Information With a Fuzzer – Leak-Fuzzer 59

4.1 Introduction 59

4.2 LeakFuzzer 63

4.2.1 Hypertesting Approach 65

4.2.2 Handling Non-Determinism 70

4.2.3 Generating {Public, Secret} Input Pairs 71

4.2.4 Handling Invalid Memory Reads 75

4.3 Evaluation and Results 86

4.3.1 Research Questions 86
4.3.2 Secure Information Flow Faults (SIFF) Benchmark Suite 87
4.3.3 Testing Environment 90
4.4 Results 91
4.4.1 RQ1: How many known insecure flows in the set of benchmarks are discovered by LeakFuzzer? 91
4.4.2 RQ2: In what proportion of runs does LeakFuzzer detect insecure flows within a standard 24-hour fuzzing budget? 92
4.4.3 RQ3: How does LeakFuzzer compare with existing approaches that could be used to detect insecure flows? . 93
4.4.4 Memory Usage Comparison 98
4.5 Threats to Validity 99
4.6 Conclusions and Future Work 100
5 Quantifying Information Leakage With a Fuzzer – NIFuzz 101
5.1 Introduction 101
5.2 Related Work 105
5.2.1 Static Analysis 106
5.2.2 Dynamic Analysis 106
5.2.3 Fuzzing Applied to Side-Channel Leakage 107
5.3 Quantifying Information Flow 108
5.3.1 Background 108
5.3.2 Estimating Conditional Mutual Information 112

5.3.3 Implicit and Explicit Flows 114
5.3.4 Direct Mappings Between Input and Output 116
5.4 Technique and its Implementation 118
5.5 Input Structure 119
5.5.1 Explicit Inputs, Stack Memory and Heap Memory Secrets 119
5.6 Mutation Phases 123
5.6.1 Explore Stage (leak discovery) 124
5.6.2 Exploit Stage (leak quantification) 126
5.6.3 Bitflips 127
5.6.4 Extending Secret Inputs - after bitflips 131
5.7 Uniform Sampling 133
5.8 Calculating QIF 135
5.8.1 Estimated Conditional Mutual Information 135
5.8.2 Channel Capacity Lower-Bound 140
5.8.3 Estimated Most Bits Leaked Directly to Output 140
5.9 Evaluation 142
5.9.1 Experimental Results 144
5.10 Threats to Validity 148
5.11 Conclusion 149

6 Improving the Exploration Efficiency of Grey-Box Fuzzing – PrescientFuzz 151

6.1 Introduction 151

6.2 Generalisable Techniques in Fuzzing 153
6.3 Our Approach 156
6.3.1 Background Concepts 160
6.3.2 Direct Uncovered Neighbours 163
6.3.3 Reachable Blocks 165
6.3.4 Rarity Weighting 166
6.3.5 Depth Weighting 168
6.3.6 Weighted Fuzzer Corpus Scheduling 170
6.4 Implementation 172
6.5 CFG File Parsing and CFG Reconstruction 175
6.6 General Fuzzer Implementation 178
6.7 Evaluation 178
6.8 Discussion 184
6.9 Impact 188
6.10 Threats to Validity 189
6.11 Conclusion 189

7 Future Work **191**
7.1 Applying the Principles of Data-Flow Guided Fuzzing to Hypercoverage 191
7.2 New Targets for NIFuzz 194
7.3 Improving Fuzzer Corpus Diversity 195

8 Conclusions **196**

Chapter 1

Introduction

This thesis is on the application of fuzzing, a software testing technique, to the detection of confidential information leaks—that is, the unintentional disclosure of confidential information—in software systems.

Privacy is becoming an increasingly important topic in the eyes of the general public, and company data breaches are part of the reason. While these are often caused by poor configuration of software systems, they can also be caused by programming errors in the software itself. Most famously such an error led to the 2014 Heartbleed bug, which affected two-thirds of the world's web servers and resulted in one of the most visited websites at the time – Yahoo – leaking user passwords [1].

The aforementioned bug reveals information due to out-of-bounds memory accesses; a form of undefined behaviour in the C programming language.

[1] https://archive.is/rHCR6

Much work has gone into detecting different types of undefined behaviour; with the emphasis gradually shifting right. Earlier tools such as Valgrind's memcheck [93] incur large overheads (10x or more); the LLVM sanitizers [107, 116] reduce this (to 2-4x); but none are suitable for running in production. GWP-ASan [110] is a modified version of AddressSanitizer which uses sampling to reduce overhead to a level that allows it to be run in production; and Arm's MTE (memory tagging extension) provides similar memory access bound-checking at the hardware level, which again can be ran in production. Whilst the programming errors can be detected by these tools, they do not provide any diagnosis of information leaks – the presence of which increases the impact of a vulnerability.

Other information leaks are due to coding errors that produce well-defined behaviour, and can only be detected using information flow control techniques. This thesis introduces new approaches and corresponding implementations that can detect leaks from both defined and undefined behaviours, and provide diagnostics to aid in their repair. Up until now, the state-of-the-art techniques for detecting information leaks either failed to computationally scale to large real-world software due to relying on formal verification; or lacked sufficient automation to effectively explore the behaviours of complex software.

Fuzzing is an automated software testing technique that relies on generating and executing vast quantities of testcases to try and find program errors that result in crashes or hangs. Hypertesting is a testing approach that can detect

violations of hyperproperties, of which confidentiality of secret information can be considered one. Despite having been proposed in 2015 [64], there has not been much uptake of hypertesting; and one of the key contributions of this thesis is the adaptation of fuzzers to perform hypertesting in chapters 4 and 5.

The underpinning theory and implementation of the first fuzzer capable of detecting input-output information leaks is documented in chapter 4; and an extension of this approach that also provides estimates for the quantity of confidential information leaked is in chapter 5. Chapter 4 introduces the new SIFF (Secure Information Flow Faults) benchmark suite, made up of a range of 3 artificial and 9 real-world programs containing information leaks; these range in size from 82 LoC (lines of code) to over 905kLoC. Chapter 5 also introduces a novel derivation for the calculation of conditional mutual information, which makes reasonable estimation of this quantity possible through testing alone. Finally, chapter 6 introduces a novel feedback mechanism and approach to scheduling the selection of inputs for mutation in grey-box fuzzers. An implementation of this approach is created and evaluated, and at the time of writing is the best performing publicly-available grey-box fuzzer as measured by rate of program exploration on a well-known fuzzing benchmark suite, FuzzBench.

The three chapters 4-6 that make up the body of thesis take the form of three papers, and as such are self-contained enough to be read independently if so desired. The background and related work chapters, 2 and 3 respectively,

introduce the relevant concepts and literature in detail, and should be sufficient to understand the entire thesis without requiring additional outside reading.

1.1 Thesis Structure

This thesis is structured as follows:

- Background – What even is *information flow* and *fuzzing*?
- Related Work – What does the existing work have to say about it?
- Detecting instances of confidential information leaks with a fuzzer – LeakFuzzer
- Quantifying leakage of confidential information with a fuzzer – NI-Fuzz
- Improving the exploration efficiency of grey-box fuzzers – Prescient-Fuzz
- Future Work and Conclusions

Chapter 2

Background

2.1 High-Level Introduction

This section provides a high-level introduction to *information flow control* and *fuzzing*; the two topics at the core of the thesis.

2.1.1 Information Flow Control

Information flow can at its simplest level be described with the following program:

```
1 int x = read() // read user input
2 int y = x
```

Here, a value is read into the variable x, this variable then contains *information* in the form of an integer. Then, an integer variable y is declared and assigned the value of x. In this program, there is an information flow from *user input* to x, and one further flow from x to y. At a macro level, we could

also say that there is a flow from *user input* to y.

Information flow control, as the name suggests, is the controlling (or restriction) of information flow. In order for control of information flow to be useful, we also need some form of policy that defines *how* information is allowed to flow; this is called a *security policy*. A typical security policy defines how information can flow from program inputs to program outputs. Note that in the prior example program above there is no explicit output. A useful security policy will also have more than one *user*, as with just one user any information provided to the program must already be known to them (who also provided said information). A *user* may not necessarily take the form of a human providing input, it could instead be a computer program, I/O peripheral, or the operating system kernel. However for the sake of simplicity and relatability let us consider an example program with two human users, user_1 who can run the program and read the output, and another user_2 who provides input but cannot read the output:

```
1 user_2.name = read()
2
3 if user_2.name == "Alice":
4     print("Hello␣" + user_2.name) // "Hello␣Alice"
5 elif user_2.name == "Bob":
6     print("Hello␣there")
7 else:
8     print("Hello")
```

A security policy for this program states that information should not flow from user_2 (providing input) to user_1 who can read the program output. This rule could more precisely refer directly to user_2's name, which in this simple example is the only information belonging to user_2.

The program violates this security policy, as `user_1` can infer, directly or indirectly, information about `user_2`. In the case that the provided name of `user_2` is "Alice", the program output *explicitly* reveals the name on line 4 in the output "hello **Alice**". This clearly violates the security policy, and it can therefore be said that the program *leaks* information (relative to our defined security policy). We call this *direct* disclosure of information an *explicit flow*.

Perhaps less obviously, the program also reveals the name provided by `user_2` in the case that it is "Bob". For any name other than "Alice" or "Bob", the program simply outputs "Hello", and we have seen that Alice's name is explicitly leaked in the output. However, for the name "Bob", the program outputs "Hello there" instead. It is therefore possible for `user_1` to distinguish when the name "Bob" was input rather than any other name based on this differing output. As the exact information contained in the input from `user_2` was not disclosed to `user_1`, but is instead *implied* by the differing output, we call this an *implicit flow*.

This property — the *non-interference property* — whereby information input by one set of users should not be observable by another set of users (in our security policy each set of users contained only one user), was first defined in Goguen and Meseguer's 1982 paper [46] with respect to an operating system as:

> *One group of users, using a certain set of commands, is* noninterfering *with another group of users if what the first group does with those commands has no effect on what the second group of users can see.*

Using this definition, it is possible to demonstrate that a program is *interfering* (or violates a non-interference based security policy) by finding a pair of inputs that can be provided by one user that result in differing observable outputs for another user. A program that does not have any instances of interference is said to *satisfy* the *non-interference property*. And the disclosure of information in a way that violates the security policy will be referred to as *information leakage* for the sake of brevity.

Much of the existing older work on information flow control focuses on formal *verification* approaches. For example, using statically-typed programming languages with built in support for security policies that can be verified for correctness during compilation [124, 90, 91]; generally this is best done while writing the software rather than trying to apply it retroactively. Another approach involves using model checking to either prove that no such non-interfering pairs of inputs exist, or the inverse [59]. Unlike typing, model checking can be done without needing to modify the program source code; however, it does require significant computational resources, and programs with many branches and / or loops struggle with the *path explosion* issue. In practice, this limits the size and complexity of programs that can be verified by model checking; in Chapter 4, we evaluate using this approach and see that in one case a 439 line C program cannot be verified within 24 hours. These approaches are particularly attractive as they provide concrete guarantees of a given program's obedience of a security policy. They do however have their downsides; the securely typed programming languages

are not well supported and not widely used in industry, and model checking suffers from scalability issues that restrict the size of programs that can be analysed.

In software testing, the counterpart to formal verification is *validation*. *Validation* is generally considered to be comprised of several subcategories, but the one of most relevance to this thesis is software testing. Following on from the revelation that interference can be proven with just a single pair of inputs, we can see that software testing can be used to find one of these pairs (if indeed any exist). There have also been efforts to quantify information leakage by calculating mutual information between sets of inputs and their corresponding observable outputs [29, 30]. These inputs must be generated by the user when using these tools, and the documentation for the Java-based implementation of the latter tool, `LeakWatch`, uses a random number generator to produce these in the example. A testing-based approach can fail to find information leaks that do exist, however unlike formal verification approaches, it is very difficult to decide when to give up trying. As Dijkstra said "Program testing can be used to show the presence of bugs, but never to show their absence". Of course, it can be argued that for sufficiently small input spaces it is possible to test exhaustively; however, the set of trivial programs to which this applies are of limited interestingness [1].

This idea of generating program inputs, running them to obtain the outputs, and then comparing these in order to determine whether there is interfer-

[1] `https://archive.is/4fQL2` provides one interesting example

ence is a highly scalable approach. By this, I mean that it should scale to any program size or complexity — provided that the program can be run in reasonable time — in contrast to the *formal verification* approaches. It also requires no modification to the programs that will be tested.

This thesis explores information flow control using a testing-based approach, and harnesses a leading testing methodology, *fuzzing*, in order to do so. Fuzzing has already proven itself as a viable tool in academia, industry and the open source software worlds; with a particularly large initiative – OSS-Fuzz [108] – having discovered, and led to the fixing of, $> 10,000$ vulnerabilities and $> 36,000$ bugs across $> 1,000$ open source projects [52].

2.1.2 Fuzzing

Fuzzing is a testing technique that has been applied to software since at least 1990 [87], and the general concept has likely existed under different names for much longer. At its simplest level, fuzzing is the process of generating random 'semi-correct' inputs and feeding them into a program with the intention of discovering which ones trigger bugs. Compared to other testing approaches, the majority of modern implementations have no knowledge of whether the program output is correct as per the specification of the program, or even what a correct output looks like. Instead, the only error class of which they are aware is crashes. This oracle may seem too simple to detect errors in professionally-written software, however for programs written in memory-unsafe languages such as C/C++, this approach can discover

bugs that would not only cause inconvenience to users, but may also be exploitable by an attacker.

It is possible to use `assert` statements to trigger crashes on certain developer decided conditions. For example, one might use `assert`s to ensure that a program does not reach an invalid state, or that a certain action has been completed before beginning another. See the example bank account manager below:

```
1  unsigned int accountBalance = 0;
2
3  void deposit(unsigned int amount) {
4      accountBalance += amount;
5  }
6
7  void withdraw(unsigned int amount) {
8      accountBalance -= amount;
9  }
10
11 int main(void) {
12     unsigned int depositAmount, withdrawalAmount;
13     scanf("%u␣%u", &depositAmount, &withdrawalAmount);
14     deposit(depositAmount);
15     unsigned int startBalance = accountBalance;
16     withdraw(withdrawalAmount);
17     printf("new␣balance:␣%u\n", accountBalance);
18     // No matter what we withdraw, we should never
19     // end up with more than we started with...
20     assert(accountBalance <= startBalance);
21 }
```

If we deposit 400, and withdraw 500, we would expect the operation to fail. This is of course poorly programmed; and worse, written in C, so naturally the compiler adds no checks to numerical operations. The output on line 17 is "`new balance: 4294967196`" on my x64 machine due to the operation $400-500$ resulting in an underflow. If we were to use a fuzzer to generate the

`depositAmount` and `withdrawAmount` inputs, it would no doubt quickly fail the assertion on line 20 and report the error.

The strategy can also be used to compare different implementations of the same specification; for example decoding a JPEG image using two different libraries, and then `assert`ing that the two resulting raw buffers in memory match exactly. Applying this strategy to the *information flow control* domain, one could run the same function with two different inputs and `assert` that the resulting outputs do not differ from the point of view of another user (i.e. the non-interference property holds).

Given the long history of fuzzing, the approach has naturally evolved, and there are now a wide variety of monolithic software systems called *fuzzers* that orchestrate all of the input generation, process spawning and crash handling elements of the process. Probably the largest advance in the realm of fuzzers was the addition of coverage feedback in order to encourage the fuzzing campaign to explore as much program functionality as possible. Note that the term *fuzzing campaign* is the name given to the process of running a fuzzer from start to finish – these typically last for hours or days, and can test billions of inputs. Using coverage to guide a fuzzing campaign is simple; record the coverage achieved by each generated program input, and if it discovers coverage that has not been achieved by any other program input then store it. The stored inputs are selected by the fuzzer and mutated in order to generate new inputs that are likely to follow similar paths, meaning that over time deeper control flow graph structures will

be exercised. Fuzzers that make use of coverage feedback are called *grey-box* fuzzers, and those that do not use any coverage feedback are *black-box* fuzzers [79]. There is a third class, *white-box* fuzzers, that use program analysis to help explore even more program functionality; they typically leverage symbolic execution (a *formal verification* approach) in order to do this. The white-box approach incurs a large overhead, and in many cases the much more lightweight and high throughput grey-box fuzzers can explore program functionalities faster. White-box fuzzers shine when testing programs with tough structured inputs including concepts such as checksums. Grey-box fuzzing is still preferred for the majority programs.

This ability to explore program functionality is the reason that fuzzing is a prime candidate for the automated discovery of information leakage (information flow control policy violations). Prior work on detecting information leakage has either required a large amount of manual intervention or fails to scale to large software systems, both of which are not an issue when fuzzing.

2.2 Thesis Motivation

This section aims to justify the existence of this thesis.

As touched on in the introduction, *information flow control* issues can have very serious consequences. Whilst the Heartbleed bug is likely the most widely known, it is not unique. Of particular note are those information leaks within operating system (OS) kernels such as Linux; where even seem-

ingly inconsequential leaks can weaken the KASLR (kernel address space layout randomisation) security defence, making it possible for attackers to perform malicious operations at a privilege level above that of any user account undetectably. The severity and exploitability of these kernel bugs has been demonstrated in two papers already [72, 28].

Furthermore, we are seeing more focus shifting to consumer privacy in the digital world; this is demonstrated by the increase in VPN usage in the US at home from 22.9% in 2019 to 79% in 2022 [123], as well as the introduction of app tracking permissions in Apple's iOS and 'ad privacy' in the Google Chrome web browser. Additionally, public concern can be seen reflected in legal directives such as the EU Privacy Directive ('Cookie Law'), the EU General Data Privacy Regulation's (GDPR) 'right to be forgotten' and California's Privacy Rights Act (CPRA). Protection of sensitive data is being treated with serious concern at both consumer and organisational levels, and software with poor information flow control implementations are a threat to this. Given the massive volume of software in use, automated techniques to detect information leaks are vital.

The existing information flow control techniques fail to scale to checking large software systems, which multi-user applications typically are; and/or require significant manual intervention to be applied to existing software. Fuzzing is an industry-proven automated approach to exploring program functionality and detecting bugs. It does, however, have a limited testing oracle, and adjusting it to detect violations of information flow control policies

is not a trivial process. It is with these points in mind, that this thesis was developed.

2.3 Grey-Box Fuzzing In-Depth

As the work in this thesis hinges around grey-box fuzzing – and in-particular AFL[2]-based grey-box fuzzers – this section will provide an in-depth overview of fuzzing.

2.3.1 What really is an input?

The most obvious software inputs for command line programs are flags, or user input from the keyboard through `stdin` (or the OS equivalent). Additionally, input could take the form of the contents of a file; this is most obvious for a command like `cat myfile.txt` where the output depends entirely on the contents (and existence) of `myfile.txt` [3]. Other *system state* could also be considered input for a program running on an OS, a subtler example may be the amount of free disk space; I would expect a different output if a file fails to get stored due to no disk space, than if it saved successfully. And this is before we have even mentioned GUIs or other computer interfaces.

From the point of view of a fuzzer, an input is at its simplest level an array of bytes; it is the job of the *fuzzing harness* to convert this array of bytes to

[2] AFL (American Fuzzy Lop) is a fuzzer first released in 2013, and is discussed shortly
[3] on unix-like OS's, `stdin` is a file too

something that can be used by the system under test (SUT). For example, the fuzzing harness may first have to write this array of bytes into a file, or deserialise it into a set of data structures used in function calls.

2.3.2 Fuzzing Harnesses

A common form for a fuzzing harness is the 'libFuzzer harness', which requires no `main` function for the SUT, but instead a definition for a function with signature `int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)`.

```
1  extern "C" int LLVMFuzzerTestOneInput(
2      const uint8_t* data,
3      size_t size
4  ) {
5      std::string buf;
6      woff2::WOFF2StringOut out(&buf);
7      out.SetMaxSize(30 * 1024 * 1024);
8      woff2::ConvertWOFF2ToTTF(data, size, &out);
9      return 0;
10 }
```

Figure 2.1: libFuzzer-type harness for the `woff2` (Web Open Font Format 2.0)

The above example is taken from `fuzzer-test-suite` [49], and is for the `woff2` (Web Open Font Format 2.0) target. The `extern "C"` is only required when using a C++ compiler. Note that the fuzzer only provides the parameters `data` and `size` (where `size` is the length of the array `data`); it is the job of the fuzzing harness to convert this into a form that can be used by the program. In the above, the target functionality is reached by calling the

`woff2::ConvertWOFF2ToTTF` function which converts one font format to another.

Despite being called the 'libFuzzer harness', it is supported by AFL [130], AFL++ [40], LibAFL [3], Honggfuzz [50] and the majority of other fuzzers written in recent years.

2.3.3 Architecture of an AFL-based Fuzzer

The original AFL was released in 2013, and is referred to in a 2023 paper as the "de facto standard for fuzzing" [41]. The following section has been written based on close examination of the publicly available source code [130]. Let's start with the name AFL, it stands for American Fuzzy Lop, which is a breed of domesticated rabbit – this does not tell us much. Figure 2.2 gives a high level overview of the components in AFL, and the subsequent fuzzers that make use of this architecture.

Input Corpus

The input corpus is a collection of inputs stored by the fuzzer. Initially this is populated using the set of *seeds* that are provided when starting the fuzzing campaign; in practice, this is a directory of files, each containing a single testcase.

In the original AFL it was called the *input queue*; and it was a queue in the literal sense, with inputs being mutated sequentially in a first-in first-out (FIFO) manner. In later fuzzers, various strategies were used to try and

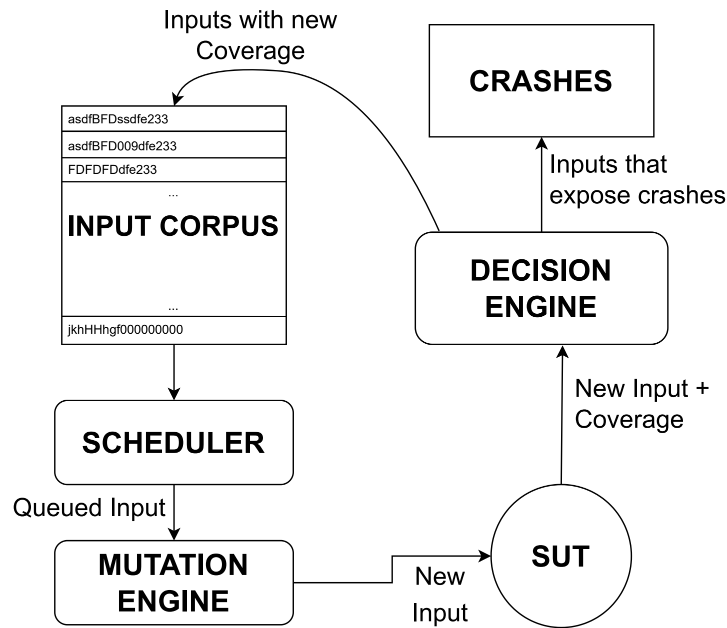


Figure 2.2: Component-level architecture diagram for AFL-like fuzzers

improve the rate of program exploration by strategically selecting the next input for mutation – this is explored in the next section on the *scheduler*. LibAFL, one of the latest iterations of the AFL architecture, now refers to the set of inputs as a `Corpus` rather than input queue; this nomenclature is preferred now that it is no longer guaranteed to be a FIFO structure.

Scheduler

The scheduler decides which input from the *corpus* to pass on to the mutation engine. The simplest schedulers are the FIFO queue from AFL, or the 'random' scheduler from LibAFL which selects an input from the corpus uniformly at random. The scheduler used by AFLFast [19] is more complex and takes into account the number of control flow graph edges covered by an input, as well as its length in bytes, execution time and the number of

times it has been mutated already.

Once the scheduler has determined what it deems to be the best input for mutation next, it passes on the input to the mutation engine.

Mutation Engine

The mutation engine takes an input that has already been tested, and mutates it in some way to create a new input [4]. The original AFL provided a set of relatively simple mutations, these are as follows:

- `bitflip 1/1`: flip a single bit in the input (e.g. `0b11111111` becomes `0b11101111`).
- `bitflip 2/1`: flip two consecutive bits in the input (e.g. `0b00000000` becomes `0b00001100`).
- `bitflip 4/1`: flip four consecutive bits in the input (e.g. `0b11111111` becomes `0b11100001`).
- `bitflip 8/8`: flip every bit in a whole byte (aligned to byte boundaries) in the input (e.g. `[0x00, 0x00]` becomes `[0xFF, 0x00]`).
- `bitflip 16/8`: flip every bit in two consecutive bytes (aligned to byte boundaries) in the input (e.g. `[0x00, 0x00]` becomes `[0xFF, 0xFF]`).
- `bitflip 32/8`: flip every bit in four consecutive bytes (aligned to byte boundaries) in the input (e.g. `[0x00, 0x00, 0x00, 0x00,`

[4] as fuzzers typically execute millions to billions of inputs during a campaign, it is inevitable that some inputs will have been generated and executed multiple times

0x00] becomes [0xFF, 0xFF, 0xFF, 0xFF, 0x00]).

- arith 8/8: add or subtract a value from a single byte in the input (e.g. [0x01, 0x02, 0x03, 0x04] could have 6 added to byte 1 to become [0x01, 0x08, 0x03, 0x04]).
- arith 16/8: add or subtract a value from two bytes in the input. This performs the arithmetic operation in either big- and little- endianness. (e.g. [0x01, 0x02, 0x03, 0x04] could have 1000 added to bytes 0-1 to become [0x04, 0xEA, 0x03, 0x04] big-endian, or [0xE9, 0x05, 0x03, 0x04] little-endian).
- arith 32/8: As above, but with 4-bytes.
- interest 8/8: replace a single byte in the input with a single-byte 'interesting' value (these are #defined as [-128, -1, 0, 1, 16, 32, 64, 100, 127] in AFL).
- interest 16/8: As above but with two-byte 'interesting' values; again both little- and big-endianness versions are tested.
- interest 32/8: As above but with four-byte 'interesting' values.
- Set a random byte to a random value.
- Delete a randomly selected slice of bytes from the input (e.g. [0x01, 0x02, 0x03, 0x04] could have the range 1-2 deleted to become [0x01, 0x04]).
- Clone a randomly selected slice of bytes and insert at a random point

in the input (e.g. `[0x01, 0x02, 0x03, 0x04]` could have bytes 2-3 cloned and inserted at 0 to become `[0x03, 0x04, 0x01, 0x02, 0x03, 0x04]`).

- Clone a randomly selected slice of bytes and overwrite another part in the input (e.g. `[0x01, 0x02, 0x03, 0x04]` could have bytes 0-1 cloned and overwrite bytes 2-3 to become `[0x01, 0x02, 0x01, 0x02]`).
- Splice two testcases together (e.g. inputs `[0x01, 0x02, 0x03, 0x04]` and `[0x05, 0x06, 0x07, 0x08]` could be spliced at positions 1 of input 1 and 1 of input 2 to become `[0x01, 0x06, 0x07, 0x08]` or at 1 of input 1 and 3 of input 2 to become `[0x01, 0x08]`).

These mutations are all relatively simple to implement, and may seem unlikely to work in practice; but when combined with the sheer volume of testcases executed (10,000 executions per second is not uncommon, and some targets can exceed 100,000 executions per second), they do work to discover new coverage. In AFL, as the fuzzing campaign goes on, more mutations are applied to the base input to create the new mutant input. By default, AFL would use a deterministic mutation strategy; first creating new inputs by applying `bitflips`, then `arith` and `interest` mutations, before moving to a `havoc` stage that applies random mutations. The big improvement seen in AFL++ [40] can be largely attributed to the deterministic mutation strategy being disabled by default, and only using the `havoc` stage. The default parameters for LibAFL allow the mutation engine to apply between 1 and 7

mutations to create new inputs – the number of mutations to apply is chosen at random.

SUT (System Under Test)

Despite having covered fuzzing harnesses in Section 2.3.2, there are more particulars to compiling the SUT for grey-box fuzzing. In particular, there is the way that the SUT is executed and also the coverage feedback instrumentation (putting the grey in grey-box).

Execution Generally, executing a program under Linux uses the `exec` family of commands. This requires the initialisation of a new process, including setting up the program runtime (and running the `_start` function), every time a new input is to be executed.

The first innovation, and one of the key features of the original AFL, was the use of a *forkserver* to mitigate this overhead. The forkserver works in the following way: the SUT is `exec`'ed to setup the runtime, but before calling the fuzzing harness function it waits for communication from the parent process (the fuzzer). When the fuzzer signals for the SUT to proceed, the SUT calls `fork`, which produces an identical copy of the SUT – this clone then executes the input provided by the fuzzer, whilst the original goes back to waiting [5]. Using `fork` eliminates the startup overhead of the SUT. Communication between the fuzzer and the SUT (including the passing of inputs) happens via a pipe; though later fuzzers instead use shared-memory for its

[5] a more in-depth discussion can be found at `https://archive.is/mEUDd`

performance advantages.

A further optimisation is so called 'persistent-mode' fuzzing; here the SUT uses a loop that waits for an input, executes it then goes back to waiting for the next input. This is even more lightweight than `fork`ing, but any modifications to the program state are not reset between inputs; hence this can only be used reliably for stateless programs.

To round out the native approaches, LibAFL offers the ability to call the fuzzing harness from directly within the fuzzer process itself. This is the most cavalier approach, as if the SUT crashes, then the fuzzer does too; a seemingly bizarre tactic given that the goal of fuzzing is to find crashes. It also provides the lowest overhead of any of the execution approaches. In practice, the issue of the fuzzer dying with the SUT is handled by the fuzzer being able to serialise its entire state to file and pass it along to a new instance of the fuzzer process (which deserialises the state to replace its own).

Finally, there are the various emulation approaches such as QEMU [13] and Nyx [105]. These emulate the entire system from a hardware level up, which is exciting as it allows for fuzzing of targets on unusual / impractical architectures such as MIPS which would typically only be found in embedded devices. These devices lack the capabilities to effectively run a fuzzing campaign on device. The emulation can also allow for the fuzzing of an OS's kernel-space, which would normally be inaccessible to a user-space program. Typically the emulation approaches will reset the program state between inputs by restoring a 'snapshot' of the entire system from before the

input was run.

Coverage Feedback In AFL, coverage feedback is provided to the fuzzer process through a 'map' in shared-memory. In practice, this map takes the form of a byte-array, where each byte corresponds to an edge in the control flow graph (in theory). Each time that an edge is covered in the SUT, the corresponding byte in the 'map' is incremented by 1. This is done by additional code that is added by a compiler pass; in practice this looks something the following:

```
1  // create a pointer to the array that makes up the coverage map,
2  // this will be mapped into the shared memory when the forkserver
3  // is initialised and will be 2^16 (65,536) bytes in size
4  uint8_t *__afl_area_ptr;
5  ...
6
7  void myFunc(int x) {
8      ...
9      uint64_t exitPCaddr = %RIP; // store the current instruction pointer
10     if (x > 6) {
11         // get the updated instruction pointer
12         uint64_t entryPCaddr = %RIP;
13
14         // XOR the instruction pointers
15         uint16_t mapOffset = (uint16_t)(exitPCaddr ^ entryPCaddr);
16         __afl_area_ptr[mapOffset] += 1;
17         // here would be the rest of the functionality
18         ...
19         exitPCaddr = %RIP;
20     } else {
21         // get the updated instruction pointer
22         uint64_t entryPCaddr = %RIP;
23
24         // XOR the instruction pointers
25         uint16_t mapOffset = (uint16_t)(exitPCaddr ^ entryPCaddr);
26         __afl_area_ptr[mapOffset] += 1;
27         // here would be the rest of the functionality
28         ...
29         exitPCaddr = %RIP;
30     }
31 }
```

Figure 2.3: A simplified demonstration of the coverage feedback that is compiled into the SUT. If we assume for simplicity that the instruction pointer value is the same as the line number, then we would see that on line 16 `__afl_area_ptr[9^11]` would be incremented, and on line 26 `__afl_area_ptr[9^18]` would be incremented.

When the SUT has indicated that it has finished executing, the fuzzer process then iterates over the map checking for non-zero bytes – these indicate edges that have been covered. The XOR approach to assigning edge indexes is unfortunately susceptible to 'collisions' whereby two or more edges map to the

same value. Some SUT have more than 2^{16} edges (such as `php_php-fuzz-parser` from FuzzBench [86], which has 123,767 edges), making collisions guaranteed; but even those with $< 2^{16}$ edges are susceptible due to the birthday problem.

There are approaches that avoid the edge collisions issue; for example, AFL++ offers a compiler pass that runs at link-time, and incrementally assigns each edge a fixed offset. This works, as at link-time all code is grouped into a single module. Additionally, LLVM now includes similar functionality in `SanitizerCoverage`, which is built into clang by default.

2.3.4 Decision Engine

Insomuch as I am aware there is no standard name for this functionality, but if I were to summarise, the 'decision engine' determines whether an input should be stored into the *input corpus*, or *crashes*. It does so based on feedback from the SUT; in AFL, whether to add an input to the *input corpus* is based solely on the coverage feedback, and whether to add to the set of *crashes* is based on the program exit code.

AFL will store an input to the *input corpus* if it finds that an edge has been covered, which has not been covered by any input up to this point in the campaign. You may also have noticed that in the prior section, the `__afl_area_ptr[mapOffset] += 1;` does not set the value to 1, but increments it. AFL will also store an input if it has covered an edge a different number of times to any input so far in the campaign. Rather than looking

for unique numbers of repetitions, of which there are 255 (excluding zero), for any edge; it instead uses 'buckets' bounded by powers of two. These are: 1, 2, 3, 4-7, 8-15, 16-31, 32-127 and 128-255. So for each edge, there could be 8 inputs added to the queue based on covering the edge different numbers of times; for example this may be seen in a loop whose iteration count depends on some part of the input.

The set of *crashes* contains inputs that have triggered an unsuccessful (non-zero on linux) exit code from the SUT. These could be caused by failing assertions within the code, by segmentation faults or other means. Similar to above, *crashes* are only stored if they cover new edges which no other *crash* has covered. Typically once an input triggering a crash is found, many thousands more will follow, so this simple sanity check for unseen edges provides some level of crash deduplication – though it is far from perfect.

Chapter 3

Related Work

The previous chapter included a high-level introduction to information flow control and fuzzing. Together, these two topics form the basis of the main goal for my thesis – that is to demonstrate how fuzzing can be used to search for information flow control policy violations (information leaks). This chapter will expand on these definitions and provide an overview of the existing literature. Firstly it dives into research in the area of information flow control, followed by the early applications of fuzzing in the area of information flow control and finally summarises work from the broader fuzzing realm in general.

3.1 Information Flow Control Background

This section provides the relevant background on the origins and early definitions in the field of information flow control.

The first attempt to formalise a security policy system for use in computing systems was the (now eponymous) model described by Bell and LaPadula in a 1973 paper [12]. It formalised the concept of each piece of information having a security classification, for example 'confidential', 'secret' or 'top secret', and the existence of an ordered set of these classifications. Here a 'piece of information' may refer to a file, stream, program or any other way of encoding information. Rules are described to ensure that information does not get read or written to by unauthorised persons, this set of rules forms a *security policy*. Users can only read information that is at their classification level, or lower down in the ordered set. Users cannot write messages to those in classification levels below them; instead they can only write messages to those with the same classification level as themselves. This work was extended in a paper published by Denning in 1976 [37] which describes a lattice structure for classification levels. This brings the advantage that several users can all access a common classification level below them (the greatest lower bound), and be accessed by a common classification above them (the least upper bound), but cannot access each others information; this is not possible in a linear classification structure. A visual demonstration of these classification hierarchies can be found below in figure 3.1:

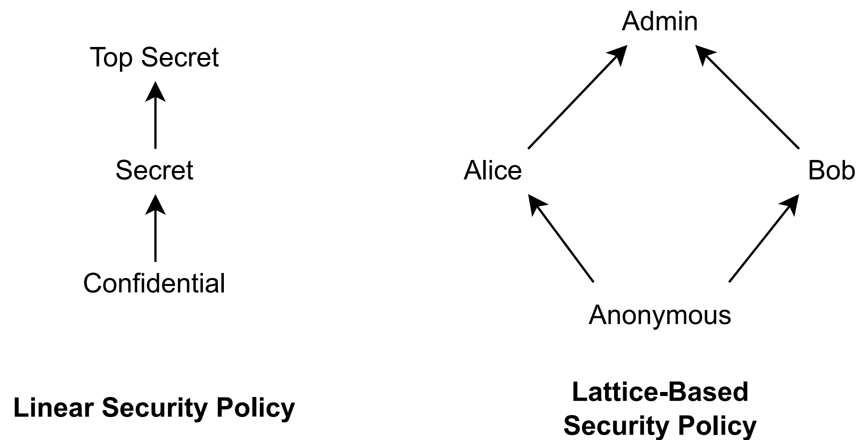


Figure 3.1: **Left:** A simple linear classification hierarchy for government documents; note that this is also technically a degenerate type of lattice called a chain lattice. **Right:** A diamond lattice-based hierarchy for an email web application. in this case a non–logged in user is *anonymous* and can view only the login page, *Alice* or *Bob* can view emails that were sent to them, and *Admin* can view all pages and emails.

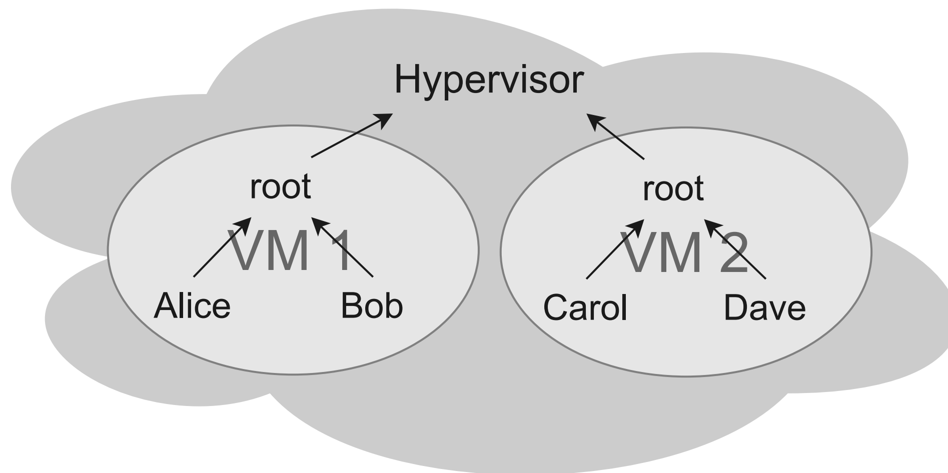


Figure 3.2: A simplified access control policy diagram for two isolated virtual machines running under a hypervisor; a common architecture for cloud computing services. Here the arrow points represent allowed directions for flow of information. Note that both VM1 and VM2 have a 'root' user, though these are separated and no information can flow between them.

As defined previously, the non-interference principle introduced in a 1982 paper by Goguen and Meseguer [46] states that one group of users is *non-*

interfering with another group of users, for a set of commands, if what the first group of users does with those commands has no effect on what the second group of users can see. The idea of non-interference is taken one step further in a 1990 paper [128] by Wittbold and Johnson, which introduces non-deducibility on strategies. This intuitively states that there is no information flow violation (information leakage) in a program, if there is no sequence of inputs where it is possible to eliminate any possible input values from the domain of high classification inputs by observing only the low classification output. In practice these definitions are strong enough to deal with most issues that are interesting when analysing programs for information flow policy violations or *information leaks* as I prefer referring to them as.

One approach for proving the presence of an information flow control violation is using a technique referred to as *self-composition* [11]. In this approach, a program is composed with itself in order to create a single program that is now equivalent to two executions of the original program. The two internal copies of the program take the same unclassified inputs, but each takes a different set of secret inputs; this macro program can then be analysed using one of the approaches described in the following subsections, in order to determine whether there exists any set of inputs (one unclassified, and two different classified) that result in an observable difference in program output.

A key difference to distinguish here is between proving the presence of in-

formation leakage, and measuring the amount of information leakage. Verifying the presence of information leakage can be done by providing a single pair of tests (hypertest) that violate the security policy. Whereas measuring true information leakage is much trickier and must be done by calculating the mutual information between secret inputs and non-secret outputs; meaning that the complete sets of inputs and outputs must be calculated. Approaches exist to estimate information leakage, and are discussed in the Statistical Approaches section (3.3.4).

3.2 Formal Verification Methods for Information Flow Control

Based on the above concepts of non-interference and non-deducibility, it is possible to statically analyse a program to decide whether there are any possible information leaks. As always, doing so does require a security policy for the program which states the classification level that each input and output is at.

3.2.1 Secure Flow Typing (Language Based Security)

The first such proposed method is called 'secure flow typing', coined in the 1996 paper [125] by Volpano, Irvine and Smith. Secure flow typing has a set of axioms, from which a set of rules are built that are applied to ensure a security policy holds for programs written in a contrived language described in

the paper. These rules can be applied across every expression in a program, building a model of the information flow from the bottom up, where any part of the program that 'types' is guaranteed to be secure on information flow. If the entire program 'types' it is therefore certain that it does not violate the security policy. Other early papers in the area [124, 126, 113] build up theoretical languages to demonstrate the concept, and this work was extended in a later paper to not only test for conformance, but also quantify the amount of information flow [32]. Practical implementations have been demonstrated in real-world programming languages Haskell [24] and Java [91, 90, 23]. These implementations require external dependencies and large adaptations to work on existing code, as well as adding an extra level of complexity for developers. For the aforementioned reasons, it is not simple to validate existing code against a security policy in an automated way, and many interesting programs are written in other programming languages. This approach of using compile-time information flow control checking is a subset of 'language-based information-flow security', and a summary of research in this area can be found in a highly cited paper bearing the same name [102].

Finally, it is worth noting that unlike other approaches, these language-based ones are preventing rather than detecting information leakage.

```
1  // Declare the checkPassword method
2  public static boolean checkPassword(
3      final String{low} guess,
4      final String{high} password
5  ):{low} {
6      // Compare the guess (low) with the password (high)
7      boolean{low} result = password.equals(guess);
8
9      return result;
10 }
```

Figure 3.3: An example password checking program written in jif – a Java-based secure flow typing language.

3.2.2 Symbolic Execution & Model Checking

Symbolic execution is a technique that has been applied to computer programs since the 1970s [66, 65, 21]. It involves converting all variables into symbolic values which represent all possible values that the variable could contain using constraint sets. The program is interpreted, and at each operation the symbolic values are updated to reflect the new possible values that they could be at that execution point. At each conditional branch in the program, the calculation of constraints must fork in order to evaluate both possible outcomes, which leads to an issue called 'path explosion' that limits symbolic execution in practice to programs of a certain complexity. Symbolic execution can be used to count the number of unique outputs, and thus for deterministic programs quantify the channel capacity (the amount of information that can be transmitted in a single program execution); this technique was demonstrated in a 2012 paper [100]. Symbolic execution has seen industry adoption [36, 34] for bug finding and has been actively de-

veloped, so that currently, engines exist to operate on LLVM bytecode [25], native binaries [27] and Java virtual machine bytecode [96].

A formal constraint-based model checking approach can be extended from symbolic execution's model, which can be used to assert that constraints hold for all possible executions of a program. `CBMC` (C Bounded Model Checker) is one such tool that can do this, and a brief description of the architecture can be found in the short paper 'CBMC – C Bounded Model Checker' [70]. It can therefore be applied to the task of searching for the presence of information leaks by asserting that there exist no two executions which differ on low classified outputs, but are identical on low classified inputs (and vary on high classified inputs); as was done in a 2016 paper by Malacaria et al. [78].

Constraint-based model checking can also be used to quantify information flow. This involves creating a SAT model of the program in question, and enumerating the number of unique outputs and their corresponding secret inputs, thus yielding the channel capacity. The issue with model counting is that it requires calculating every possible input output pair; one solution demonstrated by Heusser and Malacaria [59] is to transform the question from 'how much does a program leak?' to 'does this program leak more than `k` bits?'. It is much easier for the model checker to show this is false for low values of `k`. Biondi et al. produced a paper [15] which uses model checking with an approximate model counter that allows for analysis of larger and more complex programs. The paper demonstrates how the OpenSSL 'Heartbleed' bug can be detected using this technique, however there are serious

caveats in that a large amount of program rewriting is needed to do so. A similar hybrid approach is offered as a plugin for the Frama-C framework [10] that works for C programs.

3.2.3 Taint Analysis

Taint analysis is a technique that observes whether information input at some *source* affects the output at a *sink*. It does so by tracking assignments from tainted variables (that then become tainted themselves) and execution path differences that depend on tainted variables. Discovering information flow control violations is done by tracking taint throughout the program from a *source* and seeing if any *sinks* are tainted in such a way as violates the security policy. There are two main flavours: static taint analysis [129] and dynamic taint analysis [106]. As a result, taint analysis can fall under the static or dynamic categories discussed in this chapter, depending on the implementation.

The approach is susceptible to so called 'under-taint' in which code or data that is influenced by a tainted value is not marked as tainted. An example of where this can occur is in *implicit* information flows, where the tainted value affects control flow, but is not directly copied to a variable that is disclosed at a source. Being too liberal with taint flow can lead to the inverse problem – over-tainting – where code or data is marked as being affected by a tainted value when in practice it is actually not. One advantage of taint analysis over black-box based approaches is that it allows exact tracking of where

tainted data came from and as such can provide good feedback to developers in hunting for the logic bug. In contrast to symbolic execution, it has also proven to be very effective at detecting leakage in large systems as can be evidenced by the body of work targeting Android applications [8, 129, 44].

3.3 Dynamic Methods for Information Flow Control

The highly theoretical and proof based area of information flow control has seen less work on dynamic (generally testing-based) approaches published. Nonetheless there are some which do exactly such.

3.3.1 Assertion based Information Flow Policy Enforcement

One line of work that aimed to enforce general security policies includes a 2000 paper [39] that instrumented code at compile time with assertions that would cause the program to return early if a policy was violated. These policies were to be expressed in terms of finite state machines (FSMs) which were built into the compiled program. The states of these FSMs are updated at runtime, and at each potential transition there is an assertion to ensure it will lead to a valid state, otherwise the program will return early. The developed tool, `SASI` (Security Automata SFI Implementation), was demonstrated on both Java and C code. Despite the aim of enforcing general security policies, the authors make particular reference to constraining data-flow.

Whilst this approach is not describing a testing based approach to discovering information leaks, it can detect invalid data-flows as specified by a policy. Furthermore it can be combined with any testing approach (such as random testing) in order to search for these data-flows. It is worth noting that `SASI` can be used to detect explicit flows, but not implicit flows and this was discussed in a 2001 paper [104].

3.3.2 Hypertesting

Program properties which can be observed in a single execution are known as *trace properties*. For a program simulating a bank account, example trace properties might be that: the account balance never goes below 0; or, an appropriate error is communicated if a withdrawal is attempted that is greater than the current balance. If we can find a single input (action from a given start state) that results in a negative balance, then the first property can be shown to not hold; and if we can find a single input where an attempted withdrawal larger than the account balance results in a silent error, then the second property can be shown to not hold.

Program properties which require multiple executions to be observed are known as *hyperproperties*. Non-interference is a hyperproperty, as it requires two input-output pairs in order to demonstrate that it does not hold.

The recognition that non-interference is formally a hyperproperty was published in 2008 by Clarkson and Schneider [33]. They distinguished properties of single program executions, such as non-termination, from security

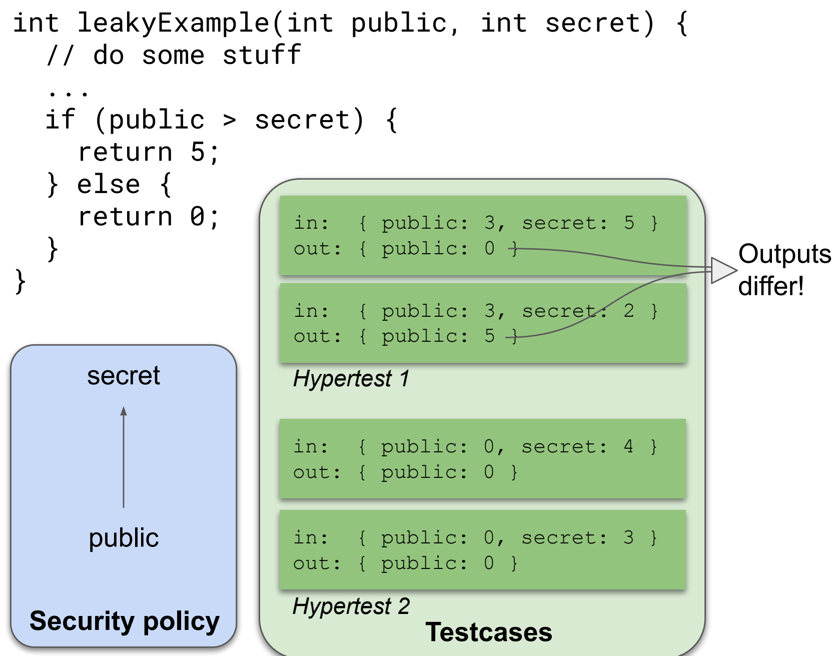


Figure 3.4: Synthetic code example with an accompanying security policy and two hypertests, one exposing a violation of non-interference and the other failing to do so

properties such as non-interference or bounds on the amount of confidential information revealed. These latter formal properties must necessarily be expressed in terms of *sets* of executions. As explained in Section 2, our non-interference property is expressed as a universal quantification over *pairs* of program executions. So a failure of the non-interference property must need a pair of tests, i.e. a hypertest, to witness a fault. Figure 3.4 illustrates the hypertest concept.

In Figure 3.4 a function, `leakyExample` is paired with a security policy where `secret` has a higher classification than `public`. The input parameter names are aligned with the security policy names, and the return value of `leakyExample` is visible to the public. Hypertest 1 has two inputs; the first has $\langle$public = 3, secret = 5$\rangle$, the second has $\langle$public = 3, secret = 2$\rangle$. Note

that only the value of `secret` changes between the two inputs. However, the (public) output values of the function `leakyExample` are different. The forgoing is a single hypertest (pair of tests in this case) that exposes a violation of the non-interference property by the program/policy pair. Not every hypertest exposes a violation as shown by hypertest 2 in Figure 3.4.

Recent papers on HyperGI, a technique that detects insecure flows and repairs them, also use hypertests [84, 85]. However, they do not have a method for automatic hypertest generation.

3.3.3 Self-Composition

Self-composition [11], as the name suggests, involves composing a program with itself in such a way as to provide the two executions with the same *Low* inputs but differing *High* (secret) inputs; the *Low* outputs can then be compared to check that they match (indicating non-interference). Self-composition is useful as it can convert a hypertest into a single test. See the following example where the input parameter `public` is classified as *Low* and `secret` is *High*:

```
1 int isLarge(int public, int secret) {
2     // do some stuff here, does not matter what
3     ...
4     if (secret > 2) return 1; else return 0;
5 }
```

To implement the self-composition approach we would create the following wrapper function that compares the output of `isLarge` with two different `secret` values:

```
1 int selfComposedIsLarge(int pub, int sec1, int sec2) {
2     assert(isLarge(pub, sec1) == isLarge(pub, sec2));
3 }
```

Any testcase for the wrapper function that causes the assertion to fail here, for example ⟨ `pub = 0, sec1 = 1, sec2 = 3` ⟩, is a witness to the insecure flow of secret information in `isLarge`. Essentially, self-composition allows a hypertest to be performed with a single testcase (given a wrapper function), thus allowing it to be used by techniques that do not natively support hypertesting.

Creating an assertion as in the above example allows for testing the (hyper)property. As a result, one approach that could be applied is property-based testing. Property-based testing is an automated testing technique, whereby instead of writing specific concrete test inputs (and expected outputs), you define general 'properties' that should always hold, and allow a tool to generate inputs in order to find testcases where the properties do not hold. For example, you could test the `abs` function with the property `abs(x) >= 0`. QuickCheck was the first property-based testing tool, and is compatible with the Haskell language [31]. Hypothesis is an equivalent tool for the testing of Python programs [77]. A property-based testing tool would be able to generate inputs and search for a violation of the assertion; this may be effective for small programs or those with easily triggered leaks, but would likely be ineffective for huge input spaces or complex programs.

3.3.4 Statistical Approaches

`LeakWatch` is a tool written by Chothia et al. [30] that measures the amount of point-to-point leakage in Java programs using a statistical approach. It is integrated as a library and requires the developer to annotate the source code to indicate values of any high classification variables (after they have been initialised) and any values that an adversary can observe (akin to low classification outputs). The `LeakWatch` command line tool then repeatedly runs the system under test in order to generate many input-output pairs, which are combined into input and output sets that can then be checked for mutual information. As this is a validation approach, the tool can infer the probability of different quantities of information leakage from each run and will continue executing the program until it reaches a specific statistical confidence level. In order to force different input values, the examples provided generate random secret inputs using Java's standard library Random or SecureRandom API's, however it is also documented that the developer can write their own input stream to allow testing of specified inputs. The `LeakWatch` tool builds on a previous tool, `LeakiEst` [29], published by the same authors, that took only input-output data from a developer produced file (i.e. no source code instrumentation library was provided) and performed a similar statistical analysis of the mutual information. The secret specification and adversary observations accept Java Objects, and these are compared using the Object's provided `toString()` implementation, which means that developer intervention is required if they wish to effec-

tively measure leakage between non-trivial objects. `LeakWatch` involves a considerable amount of developer intervention, and requires explicit marking of secret input values, in addition to being implemented only in Java.

This statistical approach to quantifying information leaks has been further explored in other papers; one such interesting tool is `HyLeak` from Biondi et al. [14], which as the name suggests takes a hybrid approach, in this case a hybrid between a formal symbolic execution like method (model checking) and a statistical approach. The intuition is that where model checking struggles with hash functions or deep loops, the tool can fall back to a statistical approach to approximate the leakage. By breaking a program down into blocks and analysing each block according to which approach seems best, and then composing the leakage for each block together, the tool can achieve a more accurate leakage estimate than pure statistical analysis, whilst also analysing programs faster than model checking. The `HyLeak` tool works on an imperative language QUAIL which is not used widely outside of academia.

3.4 Side-Channel Leakage

So far we have considered information leakage through program output, whether this be due to explicit flows, or due to the differences in output caused by implicit flows. Here, program output may refer to bytes written to file, console output, websocket or peripherals (displays or speakers for example). In fact, it is possible to leak information even without any variation

in program outputs (in the traditional sense). One can simply extend the definition of program outputs to include factors like runtime, or energy usage by the machine running the program. I prefer to term these extended set of program 'outputs' as *observables*. To demonstrate how such a leak could manifest, let's consider a simple password checking program:

```
1  password = "hello"
2  inp = read()       # Read a string from the user input
3
4  if len(inp) != len(password):
5      abort()
6
7  for i in range(0, password.length)
8      if password[i] != inp[i]:
9          abort()
10
11 # passwords match, proceed with authenticated functionality
12 ...
```

Password checkers are interesting on their own as in order to reject a password guess, they must leak some information (as the search space has been reduced by learning that the guess was incorrect); this is why quantifying information flow is interesting for certain programs. We will ignore this information disclosure here as we are focusing on side channel leakage.

Let's look at the program execution time in very much simplified terms; assuming that each program statement takes 1 second of time to execute. If we provide a program input of `"test"`, then the following path is taken (indicated by line number) $\{1, 2, 4, 5\}$ will be executed as the supplied password is not the correct length; so `"test"` takes 4 seconds to execute. If we provide `"world"` as input, then the path $\{1, 2, 4, 7, 8, 9\}$ is executed, taking 6 seconds. If instead we input `"hence"`, then we execute $\{1, 2, 4, 7, 8, 7, 8, 7, 8, 9\}$; taking

10 seconds to execute. Essentially, we observe that the program takes longer to execute as we get closer to the correct password with our guess.

In this case, a better password checking algorithm design that checked all bytes regardless of the position of the first non-match would fix the issue (or even better, compared the password's hash); however, in practice, careful implementation choices are generally the key to avoiding timing side channel leaks. Effects of differences in load speeds when reading values from RAM or CPU cache (which in itself is generally made up of multiple levels with differing speeds and sizes) can result in measurable time differences and power draws. These small differences over many runs can add up to a measurable difference.

These timing side-channel leaks have been brought to mainstream attention by recent CPU level attacks (potentially leaking information throughout the OS and between VMs) spectre [69] and meltdown [73]. Another recent CPU level attack using a different side channel, CPU clock frequency, is Hertzbleed [127]. There are many other attacks that have been demonstrated through side channels including electromagnetic emissions [4], static power draw [88] or even acoustics [5].

Side-channel leaks are a particular concern in the implementation of cryptography [115, 114]. An algorithm may be provably cryptographically secure, while a particular implementation of that algorithm may leak information about the keys; and once an attacker has a key, the cryptographic security no longer matters.

One further important note about side-channel leakage is that it can often be hardware dependent, factors such as clocks-per-instruction, memory bandwidth and CPU cache size can all impact these. As a result, not all hardware is vulnerable to the same side-channel leaks, thus an implementation that is secure on one hardware platform may not be on another. There is also literature on designing hardware that is secure against side-channel attacks [120, 55].

This thesis focuses only on the more traditional form of information leakage, where information is leaked through the 'main' channel (program outputs) as opposed to side-channels (observables).

3.5 Early Applications of Fuzzing to Side-Channel Leakage

There are several different approaches that have been applied to discovering side-channel leakage. One of the tools, `Themis` [26], uses automated reasoning in Hoare logic to search for bounded levels of side-channel leakage, returning a true / false result depending on which side of the bound the analysis lies. Another tool, `Blazer` [6], decomposes programs into smaller chunks and determines whether any variations in execution path caused by checks that are dependent on classified variables result in observables differences in execution time using abstract interpretation. Yet another technique uses neural networks to estimate the quantity of information leakage for pro-

grams [121]. This latter technique is testing based, but no general strategy is given for generating program inputs; in some cases the inputs are sampled from uniform, in another it is unspecified how inputs were generated and in one more case inputs were generated using a grey-box fuzzer, LibFuzzer [75]. Despite using a fuzzer for one demo case, I would not consider this a fuzzer-based technique. There are however four papers that do directly use fuzzing in order to detect side-channel leakage, which we will discuss now.

The first of these chronologically is the 2019 paper [94] introducing `DifFuzz`. This makes use of self-composition [11] as defined previously, which composes a program with itself in a way that creates a single program with two copies of the original program that take identical unclassified inputs and two differing secret inputs. The technique uses kelinci [62] which interfaces with AFL in a way that allows instrumented fuzzing of java programs. A fuzzing harness is used that returns the number of Java bytecode instructions executed, and this is used to guide the search towards triples (sets of one unclassified and two secret inputs) with differences in these executed instruction counts.

A 2021 follow-on paper sharing an author with `DifFuzz`, is `QFuzz` [95]. It is stated within the paper that this is an extension of `DifFuzz` which can quantify information leakage through side-channels rather than just detect its presence. Again, the setup targets Java programs, and in this instance they have 70 programs to test. They find that `QFuzz` can detect the same vulnerabilities in benchmarks that the static analysis tools `Themis` and `Blazer`

can, and `QFuzz` can detect vulnerabilities in programs that are sufficiently complex that the static analysis tools run out of time or memory. This result makes sense, as previously stated, the limitation on formal verifications (static analysis) techniques is program complexity.

A 2020 paper from different lineage describes a tool `ct-fuzz` [58]. This tool is based on AFL in its regular form, and thus targets C/C++ programs, and like `DifFuzz` uses self-composition in order to detect leakage. They do however have a much more in depth model for estimating runtime. Firstly they have a model which checks for constant time, here any divergence in execution path caused by secret inputs is treated as a leak. Secondly they have a cache model, which emulates a typical CPU cache, and any differences in terms of number of cache misses caused by secret inputs is considered a leak. To ensure that a program still has no leaks on a CPU with a different cache structure, a model of that cache would need to be made and the program tested again.

Another 2020 paper, describes an approach for detecting JIT-induced side-channels in the JVM (java virtual machine) [22]. Here JIT refers to just-in-time compilation, which is used to compile sections of code at runtime. By default the JVM interprets Java bytecode, however when a section of code is used very often, the JIT compiler compiles this bytecode into efficient machine code. This can introduce side-channels, as one particularly popular path may be JIT compiled resulting in observably faster runtimes than the less popular paths. The approach uses AFL (with the Kelinci [62] interface

to work with Java programs) to generate inputs following a particular path with the intention of running these to 'prime' the JVM to JIT compile that specific path. After priming the JVM, a random input that takes a divergent path is executed and timed. This process is repeated many times, to collect a set of runtimes for the divergent paths and for the optimised path. Then, conditional entropy is calculated between input categories and their execution time distribution. Since an input's membership of the optimised path is a binary value (either it is a member or is not), there is one bit of information that could be leaked. The tool is evaluated on a range of benchmarks including some used in the evaluation of `DifFuzz`.

My intuition is that the latter paper searching for JIT-induced side-channels has the most robust approach, as actual execution times are used. Given the large differences in IPC (instructions per cycle) of different CPU instructions, I would argue that `ct-fuzz`'s measurements for timing leakage are more robust than `DifFuzz` or `QFuzz`. As a very simple example, the Intel optimisation manual [61] suggests that on their x86 processors, the add instruction has a latency and throughput over 3 times better than the divide instruction. The approach taken by `QFuzz` and `DifFuzz` would report no leaks for the function below as both branches from the `if`-statement have the same number of bytecode instructions. In practice I would expect to see an observable difference in execution time:

```
1  int calcValue(int x, int y) {
2      int total = y;
3      for (int i = 0; i < x; i++) {
4          if (y < 5) {
5              total += x;
6          } else {
7              total /= x;
8          }
9      }
10
11     return total;
12 }
```

Outside of the work on detecting side-channel leakage using fuzzing, there is also some work from Mesecan et al. on *repairing* information leaks which makes use of fuzzers [84, 85]. In these papers, fuzzing is used to generate functional test cases; that is, the test suite for testing program functionality and *not* the hyperproperty information leakage. The method for detecting and quantifying information leakage does however make use of hypertesting, but uses a fuzzing binary search based technique as opposed to the complex genetic algorithm traditionally used in grey-box fuzzing.

3.6 Fuzzing

Having given a brief introduction to what fuzzing is and what it can detect in section 2.1.2, I will provide an abbreviated recap here before expanding on the advancements that have led the field to where it is today.

Fuzzing is a testing technique generally applied with the intention of discovering security vulnerabilities in software, dating back to at least 1990 [87]. At its core is a form of randomised testing, that is orchestrated by a tool called a

fuzzer that rapidly generates inputs and runs them as testcases; any testcases that crash are earmarked for the user to evaluate. They are generally run for a very long time, regularly at least 24 hours [68], but often even longer [89]. Fuzzing has also seen a level of industry adoption, with applications ranging from testing Android graphics drivers [56] to web browsers [122] and even high-level integration in GitLab's continuous integration services [1].

Like software testing in general, there are 2 opposing top-level approaches to building a fuzzer: a black-box approach which has no knowledge of the SUT's (System Under Test) internal structure and state, and a white-box approach which uses program analysis to improve exploration of program behaviours. An example black-box fuzzer is `Zzuf` [60]; the key advantage of the black-box approach is the sheer rate at which test cases can be generated and ran. An example white-box fuzzer is `SAGE` [45] which leverages symbolic execution to allow it to systematically test different execution paths. This process suffers from the typical limitations of symbolic execution, which limits scalability. Indeed some of the approaches based around fuzzing are incredibly complicated, take for example, `DriFuzz` [112], which uses concolic execution to figure out how to successfully initialise hardware device drivers; these drivers are executed inside of a full system emulation (using QEMU [13]).

There is one further approach; lying between black-box and white-box approaches are the aptly named grey-box approaches, the most famous imple-

[1] https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing/

mentation of which is American Fuzzy Lop [130] (AFL) and its updated relative AFL++ [40]. AFL instruments the binary for the SUT in such a way as to receive coverage information detailing which branches were run in an execution; test cases are mutated and reran, with the fuzzer storing any that lead to new coverage to a queue for further mutation. Using this queue-based mutating approach, AFL is able to explore a wide range of the SUT's behaviours without the program size and complexity constraints of a white-box fuzzer. It is worth noting that AFL includes a compiler wrapper that generates the necessary instrumentation (but can compile only C/C++), but it is also capable of fuzzing uninstrumented binaries using its built in QEMU mode. There are many extensions to AFL to support Python, Rust, JavaScript, Java, Swift, OCaml and .net [48].

A further distinction in the design of fuzzers is the method of input generation, with some opting for a grammar-based approach; meaning that test inputs are generated using a specified grammar, for example HTML when fuzzing a web browser. The alternative is the blind random based approach that makes sense for testing systems that do not take inputs of a well specified format, of course this approach could still find bugs that the grammar-based approach would, however it would likely take many magnitudes longer. AFL has support for importing a dictionary of keywords that it can use to construct its own grammar on the fly, but the default mode of operation generates random inputs without any knowledge of keywords.

As the well established benchmark of fuzzing, many fuzzing approaches

are forked from AFL. Many forks aim to improve the rate at which branches are covered or bugs discovered, for example AFLFast [19] and FairFuzz [71] which both happen to achieve gains by prioritising mutation of test cases that exercise rare branches, or AFLTurbo [118] which achieves improvements by narrowing the search space of mutations that are applied to each test case. Another approach entitled 'Hashing Fuzzing' [82] does not modify the fuzzer, but instead applies a program transformation to the SUT; in the experimentation this showed a significant improvement to coverage and the number of crashes discovered over a fixed time period. Driller [117] extends AFL to use symbolic execution to help achieve coverage of new branches when progress grinds to a halt. Another implementation leveraging symbolic execution is T-Fuzz [98] which transforms the program in such a way as to negate the entry conditions for difficult to reach branches and then uses symbolic execution to determine whether any resulting crashes could be reached in the untransformed program.

Another notable project in the grey-box fuzzing scene is LibFuzzer [75], which is maintained as part of the LLVM compiler infrastructure. As the 'lib' in the name suggests, the fuzzing engine is compiled into the target as a library. This is one step further than AFL which does require a special compilation command but only to provide instrumentation; the AFL fuzzer itself is run as an external program, and can run in black-box mode on uninstrumented executables. Given the LLVM compiler infrastructure backing LibFuzzer, it can also support more than just C and C++; in fact integra-

tions are available for Swift [7], Go[51], Rust [101] and Python [47].

There are also many specialised purpose fuzzers. One such is the `syzkaller` [54] kernel fuzzer, which was originally aimed at finding bugs in the Linux kernel but has been extended to deal with many others. Another is `nyx` [105], a grey-box fuzzer that uses a modified version of QEMU that allows very fast VM snapshotting and restoring (upto 10,000 times per second), allowing for full system fuzzing without having to worry about side effects. `Nyx` is now integrated into the `AFL++` project. `Fuzzowski` [92] is a fuzzer that specifically targets network protocols.

3.6.1 Sanitizers

LLVM also includes many *sanitizers* [53], which can be compiled into a program to cause crashes when an erroneous state is reached. These sanitizers can also be enabled in any program compiled with Clang, thus all fuzzers can benefit from them. `AddressSanitizer` detects and crashes on memory errors that are prevalent in C/C++ including use after free, heap and stack buffer overflows, memory leaks and several others; it causes a roughly 2x slowdown in execution. `ThreadSanitizer` is a tool that detects and crashes on data races, it causes a 5-15x slowdown. `MemorySanitizer` tracks uninitialised memory through program execution, and crashes if a branch is taken that depends on an uninitialised value. `UndefinedBehaviourSanitizer` detects and crashes on undefined behaviour (according to the C/C++ standards) such as integer overflows, conversion from / to floats that overflow

their new container, bitwise shifts that are out of bounds for their data type. `DataFlowSanitizer` is a taint analysis tool; unlike the other sanitizers it does not necessarily raise an error, however using assertions it can be made to crash if a particular data flow is detected.

It is worth noting that many of the issues around memory are hopefully going to slowly be alleviated by hardware checks known as *memory tagging* [111]. Each structure in memory also stores a corresponding identity tag (ARM's implementation plans to use 4-bits for this purpose [109]) which is checked by hardware each time it is accessed; if the expected tag does not match the actual tag then an error is bubbled up. For now, we still need to rely on validation and verification approaches to find these memory issues in programs.

3.7 Fuzzing for Information Leaks

There is a tool HyperFuzz (and a companion search-based software testing tool – HyperEvo) described in a 2024 paper [97] that can detect input-output information leaks, but not quantify them.

3.8 Summary

As can be seen, the area of information flow control and the techniques for detecting / preventing information leakage is dominated by formal approaches such as secure flow typing, symbolic execution and model check-

ing. Those dynamic approaches that have been proposed focus more on the assertions required and measuring techniques, and do not go into depth as to how the testing procedure would work. In order to trigger such assertions, or produce a set of input-output pairs to measure mutual information between, we must first have a set of test cases that expose the leak. One paper [121] discussed test case generation somewhat, and the majority of their test cases were generated by random testing, and for one SUT they used LibFuzzer.

Side-channel leakage is a distinct problem from program output based ('main' channel) leakage, and in this area there are four early works (one of which builds on another) that have applied fuzzing to this area with success. Side-channel leakage detection is useful for programs with implicit declassification, the obvious example of which is programs using encryption. Here the secret information is declassified and output from the program after being encrypted, however the low-privileged user cannot easily discover what the value of the secret is. Conventional information flow analysis would deem this program as leaky, so an alternative measurement for assuring security of secrets is needed. As discussed, side-channel differences can exist between hardware platforms, and some of the timing measures used such as Java Bytecode instruction counts, and cache models are far from perfect analogues of what would happen on real hardware. This platform dependency issue is not as prevalent in program output based leakage, though compiler differences such as integer and floating point widths and struct packing *can*

influence the quantity of leakage.

Finally, there is three decades worth of fuzzing work that has led to a broad variety of implementations, both general purpose and specialised. Different approaches in terms of program instrumentation, input grammars and language support also exist.

Overall, there is a clear gap whereby deploying fuzzing in a context of searching for program output based information leakage could seriously improve our abilities to detect information leakage in existing real-world programs without the program complexity constraints imposed by formal methods.

Chapter 4

Detecting Instance of Confidential Information With a Fuzzer – LeakFuzzer

4.1 Introduction

In 2015 Johannes Kinder made the case for *automated* hypertesting [64]. His motivating example was Heartbleed, the famous violation in OpenSSL of the now standard program flow security hyperproperty [33] called *non-interference* [46]. Until our creation of LeakFuzzer, Kinder's implicit challenge of automatically detecting violations of the non-interference property had not been met. LeakFuzzer is the first tool that automatically generates a set of hypertests and uses them to check that a C/C++ program together with its security policy do not violate the non-interference property. Since LeakFuzzer is a

modification of the greybox mutational feedback fuzzer AFL++ [40], it not only automatically generates new test inputs but it inherits the fuzzer's exploratory power with respect to branch coverage of the target program.

The key theoretical underpinning for LeakFuzzer is the non-interference property. In Goguen and Meseguer's 1982 paper the property was stated in a general way:

> One group of users, using a certain set of commands, is *noninterfering* with another group of users if what the first group does with those commands has no effect on what the second group of users can see [46].

There are many ways this idea can be instantiated and for LeakFuzzer we use a version of this property that frequently occurs in the secure flow literature [102]. We only observe program inputs and outputs to determine whether confidential information has been revealed. Further, the property only considers terminating inputs and assumes that each input has at most both a high security (*secret*) part and a low security (*non-secret* or *public*) part, similarly for outputs. For example, the interface through which Heartbleed is triggered has a *public* input (the packet sent by the client to the server) and a *public* output (the response sent from the server back to the client), but other parts of internal memory written to by the process during execution must be treated as *secret* as keys or other secrets may be stored there. In this work, we treat any internal memory that is written to—without the express intention of it being output to *public* output—as a *secret* input. This

(low security inputs and outputs with internal memory protected as a high input) is a common security policy in practice for multi-user programs / systems [59].

The Heartbleed bug in the OpenSSL cryptography library resulted in a buffer over-read, which exposed secret program memory information to attackers. Buffer over-reads are automatically detectable by fuzzing in combination with MemorySanitizer, now included with Clang and GCC. However, most buffer over-read bugs result in program crashes, it is rare for them to cause confidential information to be revealed to an unprivileged user. As a result, when these bugs are detected with MemorySanitizer they are usually assigned a lower priority, because the tool has no knowledge of the information leakage. LeakFuzzer is capable of detecting the insecure flow, indicating the severity of the bug. The strength of hypertesting against a security property is both that insecure flow detection is completely general, i.e. independent of the nature of the flow; and that there are no uncertainties. Either a hypertest witnesses a violation or it does not.

Existing benchmarks [49] only include a small number of programs which violate secure flow properties. To better evaluate the extended capabilities of LeakFuzzer, we collected a set of 12 C++ programs of varying sizes, from 82 to 905,264 lines of code, nine of which contain insecure flows that have been assigned CVE numbers, into a test bench we call Secure Information Flow Faults (SIFF). We have used these SUTs to compare LeakFuzzer against existing state of the art approaches to insecure flow detection. Specif-

ically we have compared LeakFuzzer against a model checking approach, that uses the C Bounded Model Checker (CBMC), and conventional AFL++ augmented in turn with two different sanitisers, MemorySanitizer (MSan) and DataFlowSanitizer (DFSan). Unlike CBMC, the mutational engine of AFL++ is nondeterministic so multiple runs were required for comparison purposes. Each setup was run 20 times for up to 24-hours (the same as FuzzBench [49]), over 400 CPU days of cumulative evaluation was performed.

LeakFuzzer found every insecure flow in every program but not with every campaign, however the majority of the 24 hour LeakFuzzer campaigns did find the sought insecure flow. In comparison to LeakFuzzer, AFL++, augmented with either sanitizer, was able to detect insecure flows in 5/12 SIFF SUTs while CBMC could detect insecure flows in 3/12 SIFF SUTs. LeakFuzzer tended to use more memory than AFL++ and less than CBMC, which exhausted the available 128GB of memory on one SUT, but there were no issues with memory exhaustion for LeakFuzzer in any experiment.

The contributions of this chapter can be summarised as:

- The creation of LeakFuzzer, the first hyperfuzzer to test against an input-output non-interference property,
- The creation of the Secure Information Flow Faults benchmark suite (SIFF), a set of varied C/C++ programs for assessing insecure flow detection tools,

- A rigorous evaluation study using SIFF that assesses the efficacy and efficiency of LeakFuzzer and compares it with three other tools: CBMC, and AFL++ augmented first with DataFlowSanitizer then with MemorySanitizer.

4.2 LeakFuzzer

Formal verification techniques for discovering insecure information flows within programs tend to suffer from an inability to scale, unlike software validation (testing) techniques. However many validation-based approaches do not solve the problem of testcase generation, which is a significant issue for programs accepting complex inputs. In comparison, fuzzing is a validation technique that is noted for its ability to handle testcase generation, achieve good program exploration and scale well to large programs. Here we present LeakFuzzer, a fuzzing-based tool developed with the goal of providing automated, scalable insecure flow detection.

LeakFuzzer is based on the popular AFL++ fuzzer [40], and as such contains all of the basic components of a standard grey-box fuzzer as used for finding regular program errors; these are shown in black in Figure 4.1. There is an *input queue* that stores 'interesting' inputs; in LeakFuzzer's case this is unmodified from AFL++, whereby an interesting input is one that previously achieved unseen program coverage and therefore should be mutated further to generate new inputs by the *mutation engine*. The *forkserver* creates instances of the system under test (SUT), feeds them the mutated inputs and

collects coverage information. Finally, the *decision engine* receives the coverage information and decides whether this input is 'interesting' and thus should be stored in the *input queue*.

The decision to extend AFL++ was made in order to take advantage of the significant effort which has gone into the development of the fuzzer infrastructure. For example, the forkserver consists of over 2,100 lines of complex code, and the shared memory coverage capturing over 900 lines at the time of writing[1]. Not only is there a large amount of code, but it is also highly optimised and implemented to take advantage of OS-specific features for Linux, MacOS and Windows. A simplistic approach that merely `exec`s the program typically achieves at least 10x less throughput [2]; making comparison with other fuzzers difficult. Finally, using a familiar platform makes it easier for other fuzzer developers to be able to pick up and extend our work to suit their purpose.

We have made a number of architectural modifications and additions in order to detect a new class of error: insecure flows. The conceptual changes are shown in green in Figure 4.1. Firstly, program output is fed back through the forkserver to the decision engine. Secondly the decision engine has been modified to process program output and store input pairs that expose insecure flows. Finally, the mutation engine has been modified as described in section 4.2.3.

[1] https://github.com/AFLplusplus/AFLplusplus

[2] https://github.com/AFLplusplus/AFLplusplus/blob/85e14cf8d137d0390bf15b67e8d972da601bb74b/instrumentation/README.persistent_mode.md#1-introduction

LeakFuzzer is able to detect a superset of the errors that AFL++ can. The hypertesting approach used in LeakFuzzer can be generalised to test for any hyperproperty, and we call a fuzzer that implements such an approach a hyperfuzzer. Note that in the following sections a simple security policy is assumed, with two classifications *secret* and *public*, where *secret* is of a higher classification than *public*.

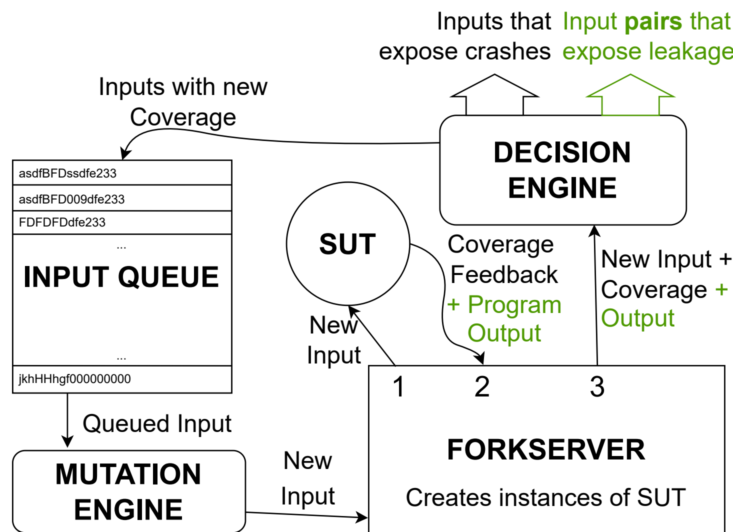


Figure 4.1: *High*-level architectural overview of a grey-box fuzzer (black), with additions for LeakFuzzer (green).

4.2.1 Hypertesting Approach

By storing a map from program inputs to outputs LeakFuzzer only needs to run each input once. This contrasts with the efficiency of methods that use self-composition (discussed previously in section 3.3.3) to achieve hypertests. Suppose that there are N distinct *High* labelled values in the test set. For each possible *Low* labelled input, self composition must in the worst case explore a space of C_2^N explicit pairs, i.e. $O(N^2)$, as opposed to LeakFuzzer's

exploration of a space of at most $O(N)$.

In order to illustrate this, let's consider this simple program which takes no public inputs, one secret input—a 2-bit unsigned integer `int` (taking values between 0 and 3 inclusive)—and returns a value that is visible to public:

```
1 int isLarge(int secret) {
2     if (secret > 2) return 1; else return 0;
3 }
```

Using the self-composition approach we could create the following wrapper function:

```
1 int selfComposedIsLarge(int secret1, int secret2) {
2     assert(isLarge(secret1) == isLarge(secret2));
3 }
```

The six possible non-repeating inputs to `selfComposedIsLarge` are `{0, 1}`, `{0, 2}`, `{0, 3}`, `{1, 2}`, `{1, 3}`, `{2, 3}`. Of these, `{0, 3}`, `{1, 3}` and `{2, 3}` cause the assertion to fail. This means that three out of the six ($\frac{1}{2}$) hypertests expose the insecure flow from *secret*. Therefore if we were to sample hypertest pairs uniformly, we would expect that two $(1 \div \frac{1}{2})$ hypertests need evaluation before we detect the flow. This means running `isLarge` four times; but it could potentially be as high as eight times in the worst case (that is, the three hypertest pairs that do not detect the flow are chosen first).

The approach taken by LeakFuzzer instead requires each input to be tested no more than once by caching the output so that it can be compared against in future without needing to be rerun. Caching outputs in full would require too much memory, hence we store 64-bit hashes of program outputs along-

side 64-bit hashes of the corresponding program public and secret inputs. Hashes are capable of colliding, however for a 64-bit hash, collisions are unlikely to effect a fuzzing campaign with 1 billion ($\approx 2^{30}$) or so inputs significantly. The XXH64 hash [2] is used due to it's extremely high throughput. Not only can we store and compare outputs for programs with large output spaces, but the comparisons can be done in a single machine code instruction due to the size of the stored hashes. This sacrifices some memory efficiency compared to naive self-composition—which requires no memory of past inputs and outputs—in exchange for improved time efficiency as each input can be executed just once.

A pseudocode listing of the insecure flow detection algorithm used by LeakFuzzer can be found in Figure 4.2, and line numbers in the following prose reference this Figure. A hash table is held in memory by LeakFuzzer, with the public input hashes used as keys, that map to values consisting of a structure containing secret input hashes, output hashes and other properties (lines 6–11). Using this hash table we can detect differences in output within pairs of testcases where public inputs are identical, and secret inputs differ. This is sufficient to infer that an insecure flow does occur.

This does not ensure that those output differences are not caused by some form of non-determinism as opposed to being deterministically influenced by the secret input. To mitigate this, LeakFuzzer stores any testcases that have resulted in a different output to other testcases with an identical public input, reruns them 100 times and directly compares the output buffers

byte-by-byte, with any testcases that give flaky (inconsistent) outputs being discarded (lines 26–31). Any testcases that pass this *flakiness test* are stored in full in the hash table value's structure (line 36), and once a pair of non-flaky, output differing testcases has been found for any public input hash then both are reported as an insecure hypertest pair and output to file (lines 38–42).

```
1  struct Input {
2    public: ByteArray
3    secret: ByteArray
4  }
5
6  struct HashValue {
7    secretInputHashes: int64[]
8    publicOutputHashes: int64[]
9    secretInputsFull: ByteArray[]
10 }
11
12 dict: HashMap<int64, HashValue> = {}
13
14 while True: # Keep going indefinitely until fuzzing campaign halted
15     generatedInput: Input = generateNewInput()
16     publicOutput: ByteArray = targetProgram.run(generatedInput)
17
18     publicInputHash: int64  = hash(generatedInput.public)
19     secretInputHash: int64  = hash(generatedInput.secret)
20     publicOutputHash: int64 = hash(publicOutput)
21
22     if publicInputHash in dict:
23         value: HashValue = dict[publicInputHash]
24         if publicOutputHash not in value.publicOutputHashes:
25             is_flaky: bool = False
26             for _ in 0..100:
27                 if targetProgram.run(generatedInput) != publicOutput:
28                     is_flaky = True
29             if not is_flaky:
30                 value.secretInputHashes.append(secretInputHash)
31                 value.publicOutputHashes.append(publicOutputHash)
32                 # Only store full input if a suspected leak is found
33                 value.secretInputsFull.append(generatedInput.secret)
34
35                 if secretInputsFull.length % 2 == 0:
36                     outputHypertestToFile(generatedInput.public,
37                                           value.secretInputsFull[-2],
38                                           value.secretInputsFull[-1])
39     else:  # publicInputHash not found in dict
40         newValue = HashValue {
41             secretInputHashes: [secretInputHash],
42             publicOutputHashes: [publicOutputHash],
43             secretInputsFull: []
44         }
45         dict[publicInputHash] = newValue
```

Figure 4.2: Pseudocode algorithm describing the hypertesting approach used by LeakFuzzer.

4.2.2 Handling Non-Determinism

As mentioned previously, a simplistic method is used for detecting non-determinism: each input in a failing hypertest is executed 100 times, and if the outputs differ between any of the executions for one of the inputs then the hypertest is discarded.

Whilst it is possible for an input that produces non-deterministic output to produce a consistent result over 100 runs, it is unlikely. Hence this method makes it very unlikely for hypertests to be incorrectly flagged as leaking due to output differences caused by non-deterministic behaviour in the SUT.

It is more likely that the approach results in false negatives; where a hypertest that actually does expose a leak is discarded due to 'flaky' outputs. This could occur if an information leak occurs simultaneously with non-determinism in the output. The following program illustrates such a case:

```
1 void leakyWithRand(int public, int secret) {
2     if (public % 1000 == 0) {
3         // example output: "1734359246992: The secret is bananas"
4         printf("%u:␣The␣secret␣is␣%d", time(), secret);
5     }
6 }
```

Here, the 'non-determinism' is caused by external state – the current time, as fetched by the `time()` call. As the time (and thus output) is likely to change during the 100 executions, this input would get flagged as 'flaky'. This results in a false negative here, as the value of `secret` is revealed in practice, but the hypertest would be discarded by LeakFuzzer due to the flakiness.

To truly overcome this, I see two potential approaches: full system emulation, or replacing all entropy sources with deterministic sources. Full system emulation can be done by QEMU, and Nyx [105] is an example of a fuzzer that does so. The strategy here is to save snapshots before execution and restore to this exact state afterwards; this ensures that any system-level state does not change between executions. Replacing all entropy sources with deterministic ones could be done by running the target program under 'hermit' [3], a fully deterministic container that was built with the intention of reproducing flaky bugs and fixing flaky tests.

4.2.3 Generating {Public, Secret} Input Pairs

The basis for LeakFuzzer, AFL++, has no awareness of the internal structure of inputs by default, and as such generates only raw byte arrays. It is up to the *fuzzing harness* to parse these byte arrays into a format that can be used by the program being tested. This is, at its core, a similar process to deserialisation.

A naive approach to deserialising a fuzzer generated input into public and secret parts could be taken if either the public or secret parts were of fixed length x. One could simply read in the first $min(\text{input.length},\ x)$ bytes needed to fill the fixed length part, and then read the remaining $max(\text{input.length} - x,\ 0)$ bytes into the other part. If both parts are of fixed length x and y bytes, then the $max(\text{input.length} - x - y,\ 0)$ bytes beyond those required can simply be discarded.

[3] `https://github.com/facebookexperimental/hermit`

In the case that both input parts can be of variable length, then the fuzzer generated input could be split in half, with the first part used as input for *Low* and the latter used as input for *High*. Alternatively, the first 1 (or more) byte(s) could be interpreted as an integer x indicating the length of the secret part of the input; the first x bytes after the length indicator would be read into the secret input, and the remaining bytes (if any) read into the public input.

These simple approaches require no adaptations to the fuzzer, however the fuzzer has no awareness of which parts of the generated input correspond to the public and secret inputs. In order to detect differences in output caused by insecure flows, inputs must only differ in their secret parts. Without awareness of which parts of the input are secret, the fuzzer cannot be certain of whether any observed differences are due to the public or secret parts of the generated inputs.

In order to overcome this difficulty, LeakFuzzer is adapted to have oversight of public and secret parts of generated inputs. This separation of input segments has several advantages. First and foremost, it allows LeakFuzzer to determine whether output differences are caused by insecure flows (assuming a fully deterministic program). Secondly, it allows for specific targeting of mutations on either part of the input (as selected from the fuzzer *input queue*).

We have discussed why a hypertest exposing an insecure flow will have identical public inputs, but differing secret inputs, so we may wish to schedule a

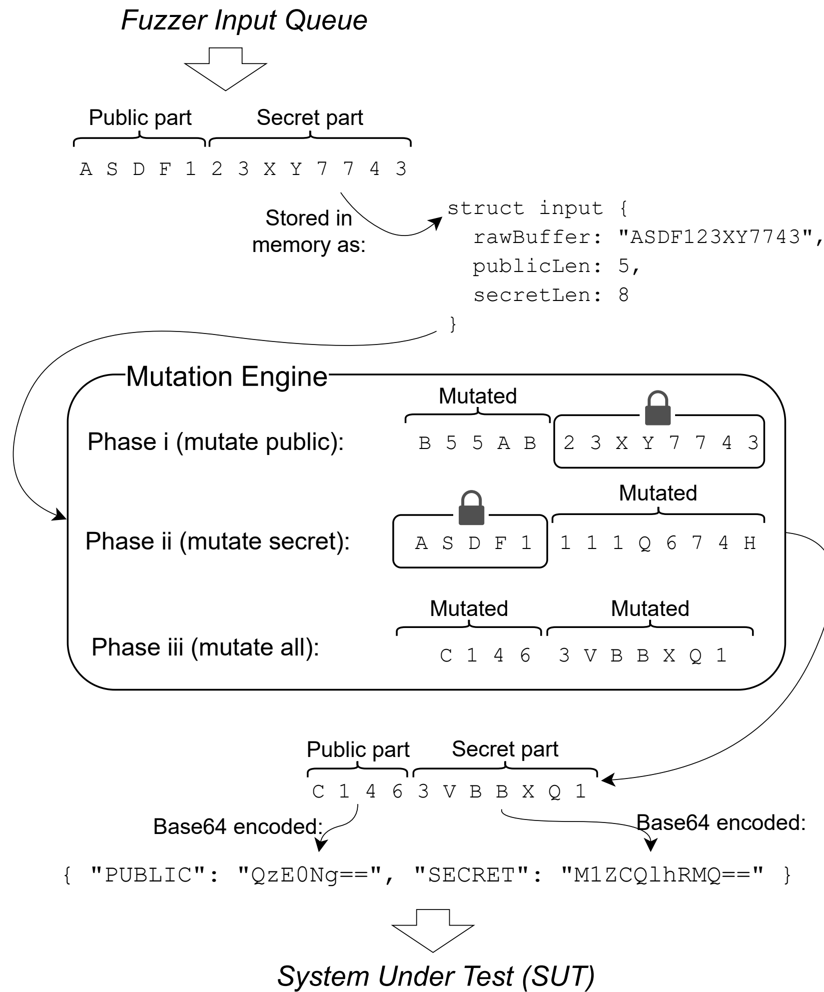


Figure 4.3: Block diagram showing the internal and external representations of inputs in LeakFuzzer.

period that mutates the secret part of the input exclusively. Figure 4.3 shows the memory structure of the stored inputs, a visualisation of the mutation phases and finally the encoded input exactly as it will be provided to the SUT (note that fuzzing harnesses must decode this back into byte arrays, and a utility function to do this is included with the LeakFuzzer source).

LeakFuzzer has three mutation phases targeting: the public part of the input, the secret part of the input, and the entire input. All three strategies are used because any part of the program input could decide whether a particular

path is chosen, and branch conditions may depend on some aspect of both the public *and* secret parts of the input. Note that the lengths of both the public and secret parts can be altered by the mutations, and this is accounted for in the fuzzer logic.

The set of available mutations that can be applied to each part of the input are unaltered from AFL++, these are:

- bitflips: flip 1, 2 or 4 consecutive input bits
- byte-flips: flipping all bits in 1, 2 or 4 consecutive input bytes
- arithmetic mutations: adding or subtracting 8-, 16-, or 32-bit values from a set of consecutive input bytes
- *interesting* value substitutions: setting 1, 2, 4 or 8 consecutive input bytes to *interesting* values such as 0, UNSIGNED_MAX, SIGNED_MAX, 1, -1 etc.
- dictionary substitutions: replacing input bytes with constant values that were collected at compile time (for example from conditional expressions) – this is effective for breaking through 'magic value' checks
- cloning: copy a random slice of the input and insert it somewhere in the input
- deletion: delete a random slice of the input
- splicing: select another input from the *input queue*, take a random slice of it; then either replace part of the current input with this slice, or insert it somewhere in the current input.

These mutations are applied in a random order, as per the default for AFL++. Initially only one mutation is applied to an input in order to generate a new

input for execution. Once a fuzzing campaign has fuzzed every queued input at least once, and has failed to discover any new coverage for at least 60 minutes, then the number of applied mutations before execution is increased by one – allowing the fuzzer to generate more diverse inputs, in the hope of busting out of the current exploration plateau.

4.2.4 Handling Invalid Memory Reads

The following subsections detail the types of memory misusage that can lead to insecure flows, and how LeakFuzzer can detect this.

How can we reliably see differences in output that are caused by uninitialised memory usage? A safe security policy might assume that internal program memory should be secret from the point of view of any user; here I refer to 'internal memory' as any memory that does not belong to a variable that would be explicitly revealed to the user under normal operating circumstances. This policy makes sense given that the layout of internal program memory is not guaranteed for languages such as C and C++; for this reason, the value just past the end of a buffer could be all zeroes, but could equally contain a secret value. Additionally, in C and C++, uninitialised memory could also contain the values of secret information previously held at that address. For these reasons, detecting that a difference in output is caused by uninitialised memory can indicate the presence of an information leak.

There exists a tool, AddressSanitizer [107], that is capable of discovering out

of bounds memory accesses and triggering a crash whenever this occurs. This is adequate for fuzzers, and allows them to discover many exploitable memory issues (even with their simple 'crash detection' oracle), however one thing that it cannot catch as of present, is copying of uninitialised memory. To explain this, I think that the following code snippet will serve better than prose:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct coord {
5      int x;
6      int y;
7  };
8
9  int main(void) {
10     struct coord myCoord;
11     myCoord.x = 4;
12
13     struct coord copy;
14     memcpy(&copy, &myCoord, sizeof(copy));
15
16     printf("(%d,␣%d)\n", copy.x, copy.y);
17 }
```

This snippet defines a `coord` (coordinate) structure which is has a integer `x` and `y` values. We define a new `coord`, `myCoord`, and initialise the `x` value, leaving the `y` value uninitialised. We then define another `coord`, `copy`, and copy the exact memory values from `myCoord` into `copy`, including the uninitialised member `myCoord.y`. One might expect that the compiler would be able to pick up on and warn about this usage of an uninitialised value, and indeed it does warn if we remove the statement `myCoord.x = 4;`. However the above snippet as it exists verbatim triggers neither compilation warnings (with `-Wall -Wextra` flags), nor Address-

Sanitizer errors at runtime.

Crucially, `myCoord.y` (and consequently `copy.y`) contains the same memory values that were previously written to this region of stack memory. If this value is output as we do with the `printf` statement at the end, then information is leaked from this memory location. This example is using stack memory, however this can occur just as easily with heap allocated memory too.

As memory can vary between runs depending on function calls and heap allocations, the leaking output may appear to be non-deterministic, when in fact a determined attacker could abuse this to reveal secrets. Therefore, finding a way to determine whether these outputs are caused by non-determinism or, in fact, a leak is a worthwhile endeavour.

Uninitialised Memory Reads

LeakFuzzer targets C and C++ programs, both of which require manual memory management. One of the consequences of this is that variables are not automatically initialised at declaration. This uninitialised variable (section of memory) then contains the values that previously occupied this section of memory; this information potentially contains secrets. One would assume that this type of error is rare, however when using `struct`s this mistake is much easier to make. It is common to declare a struct instance, and then populate the member values manually. This approach does avoid writing to each member twice (first to zero it, then secondly to set it to the final

value), however it also means that if the developer forgets to set a member then this is left uninitialised. If this uninitialised value is then output from the program, it can reveal secret information.

Out of Bounds Memory Reads

Closely related to uninitialised memory usage is out of bounds memory usage, as both can cause insecure flows due to the reading of values that were not assigned to in the current scope. Heartbleed was caused by data from out of bounds buffer access being copied to program output. An additional source of these errors that can exist in C and C++ is the use of `memcpy` with structs. This is due to *memory alignment*, which is done for performance reasons. Take, for example, the following struct:

```
1 struct instruction_t {
2     char direction;
3     int  distance
4 };
```

A tightly packed structure on a 64-bit machine (with 32-bit `int`s as per used by gcc and clang) would take up only 5 bytes: 1 byte for `direction` followed by 4 bytes for `distance`. However, assuming a preference for 4-byte memory alignment, this struct would actually take up 8 bytes: 1 byte for `direction`, 3 bytes of padding to reach a 4-byte boundary, and 4 bytes for `distance`. Naively serialising this struct to program output by using `memcpy`, the 3 bytes of uninitialised padding would also be output, potentially revealing secret information.

Solution

The *fuzzing harness* that is used to convert the raw byte-string generated by the fuzzer into a form of input accepted by the program often runs only a single input through a slice of the system. As memory pages are zeroed by the OS before being passed to the program process, it is common that early uninitialised memory reads access these zeroed regions. The longer that the program runs, the more common it is that memory is reused; and this is particularly the case in long running interactive programs such as web servers and databases. It is therefore possible that invalid memory reads that lead to confidential information being output can go unnoticed in these fuzzing harnesses, but would affect live long-running processes. Additionally, the output would change depending on the previous value stored in that area of memory and thus may appear to exhibit a form of non-determinism.

In order to allow for the detection of these potential insecure flows, LeakFuzzer uses a technique to set internal program memory to a pattern of consistent non-zero values. To do this, a portion of the generated *secret* input is used as a seed for a pseudorandom number generator, and a sequence of bytes is generated using this (in the current implementation, 8 bytes). This sequence of bytes is then replicated, until it fills the process stack memory, by a function within the initialisation phase of the fuzzing harness. Heap memory initialisation is handled by a provided wrapper for the `malloc` function, this wrapper first allocates the required memory plus an additional 8-bytes and then fills it with the repeated pseudorandom byte sequence. The addi-

tional 8-bytes cause differing outputs in the case of buffer overreads.

By seeding the pseudorandom sequence from the *secret* input, it becomes possible for LeakFuzzer to generate hypertests that can produce differing *deterministic outputs* due to invalid memory reads and thus expose the insecure flow.

Worked Example: Heartbleed

Below is an abbreviated slice of the source code containing the Heartbleed bug:

```
1  char *packet = ...; // array of bytes sent by the 'public' level user
2  uint16_t payload_length = packet[1] << 8 | packet[2];
3  ...
4
5  // Read from the payload_length field from the request packet
6  n2s(packet, payload_length);
7  ...
8
9  // buffer stores the response packet
10 unsigned char *buffer, *bp;
11
12 /* Allocate memory for the response, size is 1 byte
13  * message type, plus 2 bytes payload_length, plus
14  * payload_length, plus 16 bytes padding. Note that
15  * this buffer is never accessed out-of-bounds, instead
16  * it is 'packet' that gets overread */
17 output_buffer = OPENSSL_malloc(1 + 2 + payload_length + 16);
18
19 ...
20
21 /* copy 'payload_length' bytes from 'pl' to 'output_buffer' where:
22  * 'pl' is a pointer to the payload within 'packet' and
23  * 'output_buffer' is the buffer that will be returned to the user */
24 memcpy(output_buffer + 3, pl, payload_length);
25
26 ...
27 OPENSSL_free(buffer);
28 // send the populated buffer back to the 'public' level user
29 r = ssl3_write_bytes(
30         s, TLS1_RT_HEARTBEAT, output_buffer, 3 + payload + padding
31     );
```

The issue here is that on line 6, the value of `payload_length` is read from the *public input* `packet`; this number of bytes is then copied to the *public* program output `buffer` on line 24. The *public* user is able to set `payload_length` to a value between 0 and 65,535; and crucially, they can send a much shorter

actual payload (say 1 byte), leading to a buffer overread, the contents of which are then returned to them through *public output* on lines 29–32. By default, LeakFuzzer uses a security policy which labels any internal program memory that should not be returned to *public* as *secret*; in this particular case, the contents of the response header, payload from `packet`, and padding bytes should be returned to *public* in the response packet, but anything else should not be.

The contents of the buffer overread will be internal program memory; whatever happens to be allocated in the memory following `packet`. In a freshly brought up system, as the fuzzing harness is, it is likely that this memory has not yet been used and will contain 0s; in a long running server application, this memory could contain secrets such as login request details or other confidential information. Note that OpenSSL is a library for allowing secure communications, and as such is built in to many server-side applications. In the case of a fuzzing harness whereby internal program memory is either zeroed or does not change between executions (due to program state getting reset), sending a malformed packet that triggers the Heartbleed bug many times would not result in any observable difference in output. This is where the LeakFuzzer's aforementioned techniques for handling invalid memory reads come into play.

Sent Packet (*public* input)

0	1	2	3	4	5	6
0x01	0x00	0x09	0x74	0x65	0x73	0x74
Packet type: 1	payload_length: 9		't'	'e'	's'	't'

`packet` contents at line 1 without LeakFuzzer approach to memory seeding

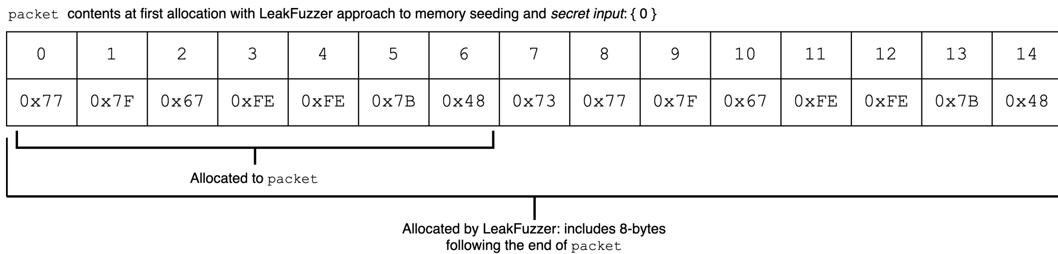
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0x01	0x00	0x09	0x74	0x65	0x73	0x74	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Allocated to `packet` (bytes 0–6) | Not yet allocated to anything (bytes 7–14)

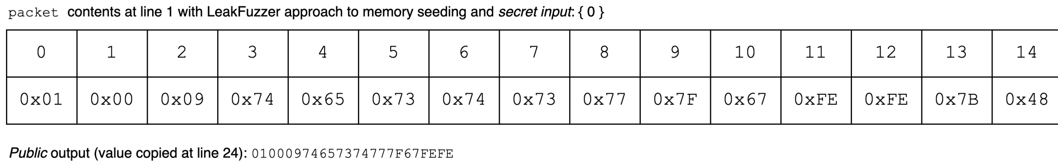
Public output (value copied at line 24): `0100097465737400000000000`

Figure 4.4: An example packet triggering Heartbleed. The top table shows the internal format of the heartbeat request packet as sent by the client. The bottom table shows the memory layout of the allocated `packet` buffer from line 1 of the code listing above; note that the memory after `packet` has not yet been allocated to anything and is thus zeroed. Finally, the bottom line of text shows the output contents sent back to the client (i.e. the output) in hex string form; no matter how many times the fuzzing harness is ran, this output will not differ.

The `packet` variable is allocated on the heap by `OPENSSL_malloc`, the OpenSSL project's custom wrapper for `malloc`. The `malloc` call within the wrapper is replaced at compilation time by LeakFuzzer, meaning that an extra 8-bytes are allocated and all of the allocated bytes are populated with values generated from the *secret* input that LeakFuzzer generated. Now when a malformed packet triggering Heartbleed is provided as *public* input, the corresponding *public* output is affected by the values of those 8-bytes due to the buffer overread. Producing a hypertest consisting of two inputs with matching *public* parts and differing *secret* parts will demonstrate the insecure flow, as the outputs now differ.


`packet` contents at first allocation with LeakFuzzer approach to memory seeding and *secret input*: { 0 }

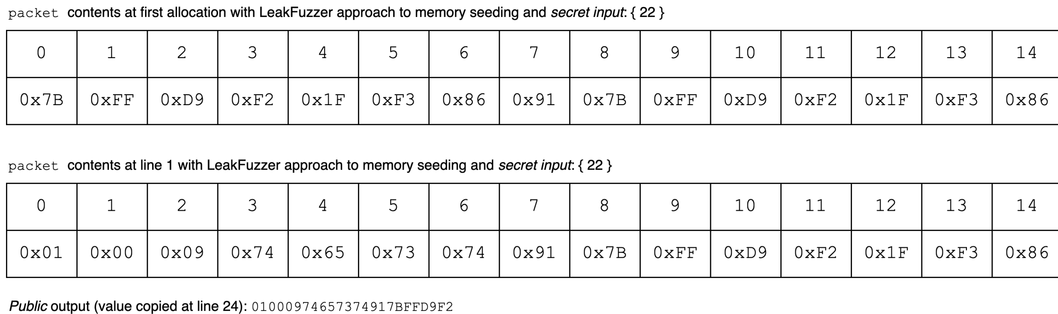
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0x77	0x7F	0x67	0xFE	0xFE	0x7B	0x48	0x73	0x77	0x7F	0x67	0xFE	0xFE	0x7B	0x48


`packet` contents at line 1 with LeakFuzzer approach to memory seeding and *secret input*: { 0 }

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0x01	0x00	0x09	0x74	0x65	0x73	0x74	0x73	0x77	0x7F	0x67	0xFE	0xFE	0x7B	0x48

Public output (value copied at line 24): `01000974657374777F67FEFE`

Figure 4.5: The top table shows the memory layout of `packet` when allocated by LeakFuzzer, note the extra 8 bytes allocated after the end of packet and the repeating 8-byte pattern that has been used to initialise the memory. The bottom table shows the memory layout after it has been populated with the contents of the sent `packet`; note that bytes 0-6 are unchanged from Figure 4.4, but 7-14 are now non-zero. The program output now differs compared to Figure 4.4; but does not change if the fuzzing harness is reran with the same *secret* input (0).


`packet` contents at first allocation with LeakFuzzer approach to memory seeding and *secret input*: { 22 }

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0x7B	0xFF	0xD9	0xF2	0x1F	0xF3	0x86	0x91	0x7B	0xFF	0xD9	0xF2	0x1F	0xF3	0x86

`packet` contents at line 1 with LeakFuzzer approach to memory seeding and *secret input*: { 22 }

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0x01	0x00	0x09	0x74	0x65	0x73	0x74	0x91	0x7B	0xFF	0xD9	0xF2	0x1F	0xF3	0x86

Public output (value copied at line 24): `01000974657374917BFFD9F2`

Figure 4.6: The top table again shows the memory layout of `packet` when allocated by LeakFuzzer, this time a different *secret* input, 22, was provided and hence a different repeating 8-byte pattern generated. The bottom table shows the memory layout after it has been populated with the contents of the sent `packet`; again bytes 0-6 are unchanged from the previous figure, but 7-14 differ. The program output differs compared to Figure 4.5 due to the different *secret* input values (22 and 0).

From the above Figures, we can see that the pair of tests:

`{ public: 01000974657374, secret: 0 }` and

`{ public: 01000974657374, secret: 22 }` deterministically pro-

duce different outputs `0100097465737477 7F67FEFE` and `01000974657374917BFFD9F2` respectively. As Figure 4.4 shows, without LeakFuzzer's approach to memory handling, we would not be able to observe any difference in output.

4.3 Evaluation and Results

Having described LeakFuzzer and how it works, in this section we evaluate how well it performs against faults that cause insecure flows in the benchmark software suite we have assembled. We assess the usual questions of efficacy and efficiency. We then evaluate how well its performance compares to a representative sample of available tools that can be used to detect insecure flows.

4.3.1 Research Questions

RQ1: How many known insecure flows in our SUTs are discovered by LeakFuzzer?

We seek to answer efficacy permissively, by the percentage of SUTs in which the known error is found in at least one of the 20 24-hour runs.

RQ2: In what proportion of runs does LeakFuzzer detect insecure flows within a standard 24-hour fuzzing budget?

Here we seek to determine the efficiency of LeakFuzzer by considering the mean time to discovery and the proportion of runs that detect leakage of confidential information within a standard fuzzing campaign budget of 24 hours.

RQ3: How efficacious and efficient is LeakFuzzer in comparison with existing practical approaches that can detect insecure flows?

We compare with three tools: MemorySanitizer, DataFlowSanitizer, and the C Bounded Model Checker (CBMC). We compare the results using the same metrics as used in RQ1 and RQ2 for all three tools. Additionally we look at measurements of memory

usage.

First we discuss the creation of the benchmark suite.

4.3.2 Secure Information Flow Faults (SIFF) Benchmark Suite

No benchmark suite for insecure flows in C and C++ was available to our knowledge. Hence we created SIFF, a secure information flow faults repository that contains programs together with fuzzing harnesses, initial fuzzing seeds and example insecure flow exposing hypertests. Information flow control issues are difficult to find in known issue databases due to the language used to describe them. Typically they are referred to as *information leaks*, however the word *leak* is overloaded and could refer to memory leaks, resource leaks or confidential information leaks, with the former two being more prevalent in C and C++ programs. In total we have included 12 distinct programs, of which 9 are taken from confirmed CVE reports and others are example programs taken from existing work. Implicit and explicit flows are represented in the program suite as well as memory management and design issues. It also has examples of different sized SUTs and a mixture of kernel and user space programs. In addition, the SUTs have variety in input types including byte strings, human readable formats and SQL queries. The programs taken from CVE reports span errors from 2007 to 2022. We have taken all of the programs containing CVEs from the work using CBMC to detect information leaks (insecure flows) [59]. The repository is publicly

available for reuse in future research.

A fuzzing harness has been constructed for each individual program. The OpenSSL-1.0.1f fuzzing harness was modified from the version contained in fuzzer-test-suite [49]. Additionally, when testing using AFL++ based systems, the PostgreSQL program uses the AFL++ Grammar Mutator [1] plugin to generate SQL statements from a provided grammar. This is because the standard mutation engine struggles to generate valid statements even when provided with many seeds.

Table 4.1: Program details for members of the Secure Information Flow Faults benchmark suite. Note that Flow Type is abbreviated using the key: E: explicit flow, I: implicit flow, M: Memory issue, PD: Program design issue.

Program Name	Flow Type	LoC	CFG Edges	CVE Number	Source
appletalk	E + M	110	15	CVE-2009-3002	[59, 67]
Banking	I + PD	150	62	-	[57]
cpuset	E + PD	82	7	CVE-2007-2875	[59]
NetworkManager	E + PD	18,185	770	CVE-2011-1943	-
OpenSSL-1.0.1f	E + M	279,466	33,262	CVE-2014-0160	[49]
Password Check	I + PD	117	63	-	[57]
PostgreSQL	E + PD	905,264	5,862	CVE-2021-3393	-
RDS	E + M	94,248	43	CVE-2019-16714	-
Reviewers	I + PD	145	86	-	[57]
sigaltstack	E + M	141	34	CVE-2009-2847	[59]
sr9700_rx_fixup	E + M	439	28	CVE-2022-26966	-
tcf_fill_node	E + M	810	169	CVE-2009-3612	[59]

All programs are written in C with the exceptions of Banking, Password Check and Reviewers; these were originally written in Java and manually converted to C++. Note that the LoC (lines of code) was computed using `cloc` on the source directory for each project; the number of reachable lines compiled into the executable as run by the fuzzer may be significantly less. The CFG Edges column above indicates the number of control flow graph

edges that were present in the compiled binary.

Program Design Errors

Of the 12 programs, six contain insecure flows caused by program design error, where a flow logic error causes the leakage. For these programs, there is explicit *secret* input, *public* input and *public* output; the bug report allows us to construct a security policy for them.

Memory Mismanagement Errors

Six of the programs contain memory related errors and all are detected by LeakFuzzer. For these we use a security policy whereby all user input is labelled *public*, all output is labelled *public*, and any other internal program memory is labelled *secret*. In order to allow us to populate the *secret* internal program memory from input, we use the techniques described in section 4.2.4. The first of these is appletalk, this is a simplified version of the original bug taken from a previous paper on measuring information leakage [59]. This reveals confidential information from stack memory within the Linux kernel, and is caused by failure to initialise all members of a struct before returning the value to userspace. OpenSSL-1.0.1f contains the infamous Heartbleed bug which revealed information by outputting internal program heap memory due to a buffer over-read. RDS (reliable datagram sockets), sigaltstack and tcf_fill_node are also taken from the Linux kernel, with information revealed from stack memory; again caused by failure to initialise all struct members. Finally, sr9700_rx_fixup is taken from a Linux

network driver which reveals information from heap memory due to buffer over-read.

4.3.3 Testing Environment

All tests were run inside Docker containers based on the Ubuntu 20.04 distribution of Linux. Fuzzing experiments were run with 10 fuzzing campaigns in parallel, each bound to a single CPU core. Model checking (CBMC) experiments were not run in parallel due to RAM space being a common bottleneck. Tests were run on dedicated servers each equipped with 2 x Intel Xeon E5-2620 v2 processors, making for a total of 12 cores (24 threads) at 2.10GHz, and 128GB RAM. As is common in the evaluation of fuzzers, each benchmark was fuzzed for 24 hours, though this time length does not guarantee that exploration progress has halted for large programs [68].

Each experimental setup/run was repeated 20 times. The nature of the mutational engine on the inputs is nondeterministic so different results may occur in different 24 hour runs.

4.4 Results

4.4.1 RQ1: How many known insecure flows in the set of benchmarks are discovered by LeakFuzzer?

Table 4.2: Results for LeakFuzzer on the SIFF benchmark suite.

SUT	% runs flow detected	flow detected time (s)	
		Mean	Std. Dev.
appletalk	100	175.23	250.49
Banking	100	1.48	1.25
cpuset	95	1.16	0.41
NetworkManager	100	228.01	141.43
OpenSSL-1.0.1f	100	455.08	266.39
Password Check	100	483.65	1,575.00
PostgreSQL	55	32,870.09	19,788.94
RDS	100	3,223.85	8,910.57
Reviewers	100	82.53	87.38
sigaltstack	100	55.42	17.65
sr9700_rx_fixup	40	16,530.35	27,442.28
tcf_fill_node	100	64.86	44.92

For all 12 programs, insecure flows were detected in at least 40% of the runs. This includes all flow types, in particular five of the six program design errors were detected in 95% or more runs. Discussion on the cause of variability follows in RQ2.

> LeakFuzzer finds every insecure flow in every program, but not in every run.

4.4.2 RQ2: In what proportion of runs does LeakFuzzer detect insecure flows within a standard 24-hour fuzzing budget?

Each run used the same set of initial seeds. As can be seen in table 4.2, even for the same program, runtime before detecting the first insecure flow varied considerably due to non-deterministic input queue and mutation selection strategies used in AFL++.

The two most complex programs as measured by lines of code—OpenSSL-1.0.1f and PostgreSQL—had the longest discovery times, as would be expected with an exploratory approach such as fuzzing. LeakFuzzer took a surprisingly long time to discover the insecure flow in sr9700_rx_fixup despite containing relatively few lines of code due to the part of the input space that triggers the bug being very small. In this particular case there were 28 edges that could be covered, and in 19 of the 20 runs 17 edges were covered; only 12 were covered in the other run. As might be expected, the insecure flow was not detected in the run that produced less coverage. As the other runs demonstrate, there is not a one-to-one relationship between coverage and the discovery of insecure flows. We see a similar lack of correlation between coverage and insecure flow discovery for both OpenSSL-1.0.1f and PostgreSQL.

It is possible that all fuzzing campaigns would have discovered insecure flows if left to run for long enough; however within the 24-hour time limit imposed here the majority still did.

4.4.3 RQ3: How does LeakFuzzer compare with existing approaches that could be used to detect insecure flows?

Both MemorySanitizer (MSan) and DataFlowSanitizer (DFSan) were evaluated using a similar fuzzing harness and input seeds. These setups were fuzzed by an unmodified version of AFL++ for 24-hours each to provide a fair comparison against LeakFuzzer. Those programs containing an insecure flow caused by a memory issue were evaluated with MSan, and those containing a program design issue with DFSan.

SUT	Sanitizer	% runs flow detected	Flow detected time (s)	
			Mean	Std. Dev
appletalk	MSan	0	-	-
banking	DFSan	0	-	-
cpuset	DFSan	100	0.12	0.11
NetworkManager	-	-	-	-
OpenSSL-1.0.1f	MSan	100	2377.47	2732.26
Password Check	DFSan	0	-	-
PostgreSQL	DFSan	0	-	-
RDS	MSan	100	548.71	152.25
Reviewers	DFSan	0	-	-
sigaltstack	MSan	100	330.78	225.23
sr9700_rx_fixup	MSan	0	-	-
tcf_fill_node	MSan	100	101.96	85.25

Table 4.3: Results for AFL++ in combination with the two sanitizers on the SIFF benchmark suite.

Memory Sanitizer (MSan)

As one of the LLVM sanitizers, MSan can be used in combination with fuzzing to detect uninitialised memory usages. For this reason, it may be able to detect a proportion of the insecure flows caused by memory mismanagement errors. It is worth noting that the presence of uninitialised memory use does not necessarily imply that the program output will be affected, thus there may not be confidential information revealed by many errors detected by MSan. In order to determine whether a crash detected by MSan 'detected' an insecure flow, the stack trace for each discovered error was manually inspected. If fixing said error would eliminate the flow then it was considered to have been 'detected' in that run. Due to repeats sharing the same executable, each unique crash (i.e. those sharing a stack trace) only needed inspecting once.

The four programs containing insecure flows caused by memory mismanagement —appletalk, OpenSSL-1.0.1f, RDS and sr9700_rx_fixup—were evaluated with MSan, as it can potentially detect these. Of this subset, errors related to the known insecure flow were discovered in OpenSSL-1.0.1f and RDS by MSan. It is likely that the error in OpenSSL-1.0.1f was discovered much more quickly by MSan than LeakFuzzer as many inputs that trigger the buffer over-read do not reach far enough in memory to result in disclosure of confidential information. LeakFuzzer instead requires that there are discernible differences in output so must find those inputs that trigger this behaviour. The error in RDS was also detected more quickly by MSan,

though not by such a significant margin.

DataFlowSanitizer (DFSan)

As it provides an implementation of taint analysis, DFSan is more closely related to information flow control than MSan. Where MSan required no modification to the fuzzing harnesses, for DFSan we added labels to the secret inputs and asserted that this label did not propagate to the program output. Any input that resulted in the assertion failing was considered to have discovered the insecure flow.

We had expected that DFSan would be unable to detect the three implicit flows due to the fact that DataFlowSanitizer tracks only *data flow* and not *control flow*; this proved to be the case in all runs. It was slightly less clear whether PostgreSQL would trigger an assertion failure due to the secret values being stored into the database before being fetched in certain queries that expose the insecure flow. As the database might write the values to disk and retrieve from disk, we expect that any taint labels are lost in this process. It is also possible that no query exposing the insecure flow was generated, however it seems unlikely that this would be the case for all 20 runs given that LeakFuzzer succeeded in 11 of its 20 runs. Whatever the cause, the results show that for the setup that we used, DFSan was unable to detect the information flow from secret input to public output. Finally, we observe that the error in cpuset is discovered very quickly by DFSan; this is a simple program, and is explored by the fuzzer very quickly.

The NetworkManager benchmark is a better candidate for detection by DF-San than by MSan, however as both require all dependencies to be compiled with sanitizer flags it was not tested. Attempts were made to resolve all dependencies. However after significant time was spent attempting to do this all the way down the stack too many issues were encountered to consider going any further. Note that the 'banking', 'password check' and 'reviewers' benchmarks required building the C++ standard library from scratch too, though this was managed in reasonable time.

C Bounded Model Checker (CBMC)

Model checking harnesses were created for each of the benchmarks to allow the model checker to verify the presence of an insecure flow. An issue was encountered in building the 'banking', 'password check' and 'reviewers' benchmarks, as these are written in C++. This is due to a known issue that CBMC parser "does not handle more modern and template-heavy C++, this means it often runs into problems with parts of the standard library" [119]. As a result, the model checker was evaluated on the 7 benchmarks written in C.

SUT	% runs flow detected	Flow detect time (s)		Exit Cause
		Mean	Std. Dev	
appletalk	100	0.29	0.10	Success
cpuset	100	1.21	0.38	Success
NetworkManager	0	-	-	Segfault
OpenSSL-1.0.1f	0	-	-	Timeout
PostgreSQL	0	-	-	Timeout
RDS	0	-	-	OOM
sigaltstack	100	0.17	0.013	Success
sr9700_rx_fixup	0	-	-	Timeout
tcf_fill_node	0	-	-	Timeout

Table 4.4: Results for CBMC on the SIFF benchmark suite.

The three of the smallest benchmarks—appletalk, cpuset and sigaltstack—finished quickly and CBMC managed to find a counterexample falsifying the assertion. In two of these cases, it was able to detect the insecure flow faster than LeakFuzzer on average. LeakFuzzer had a mean discovery time of 175.23s and 0.17s for appletalk and sigaltstack respectively, compared to 0.29s and 0.17s for CBMC.

CBMC struggled with larger programs as again a 24-hour time limit was imposed, leading to four subjects consistently exiting before the model checker was finished (marked in the table as 'timeout'). Note that tcf_fill_node timed out in our CBMC experiments, whereas it did not in the CBMC approach's original evaluation [59]. In the original paper they explain that they use a simplified version of the kernel code but do not provide their source code. We instead simply isolate the slice of code without simplifying it, which may explain why our benchmark version does not successfully complete. In the case of RDS, the 128GB of memory available on the evaluation hardware was exhausted causing an out of memory (OOM) early exit. Finally, when

attempting the verify NetworkManager, CBMC would exit due to a segmentation fault within the model checker itself.

The formal verification approach was faster in the instances where it managed to resolve however, as has been found in prior works, it fails to scale well enough to work on the larger programs.

4.4.4 Memory Usage Comparison

SUT	Mean Memory Usage (MiB)		
	LeakFuzzer	AFL++	CBMC
appletalk	2,671.68	28.73	19.13
banking	22,699.78	28.80	-
cpuset	1,926.38	28.74	12.22
NetworkManager	1,622.63	28.27	1,633.14
OpenSSL-1.0.1f	3,196.45	37.75	66,685.88
Password Check	603.52	29.03	-
PostgreSQL	710.54	4.27	7,734.69
RDS	7,433.65	32.30	127,203.04
Reviewers	423.30	36.76	-
sigaltstack	4,927.68	30.50	19.06
sr9700_rx_fixup	5,949.39	205.25	22,477.16
tcf_fill_node	2,928.99	30.50	29,630.58

Table 4.5: Table of results for Memory Usage

One of the concerns with storing information about every tested input in LeakFuzzer as described in section 4.2.1 is the potential memory usage. As can be seen, over the 24 hour runs, memory usage of LeakFuzzer is certainly much higher than AFL++ with the sanitizers, but typically lower than CBMC in the long running test cases. None of the 200 LeakFuzzer runs ran out of memory on the evaluation machine equipped with 128GB RAM.

LeakFuzzer was able to detect the insecure flow in all 12 of the programs in some proportion of the 24-hour runs. The combination of AFL++ with the sanitizers was able to detect insecure flows in five subjects, and CBMC was able to detect flows in just three subjects. Of the 12 programs, insecure flows were detected by all methods in just two: cpuset and sigaltstack. Additionally, LeakFuzzer has been shown to use more memory than AFL++ as expected, however generally less than CBMC, and there were no issues with memory exhaustion in our tests.

4.5 Threats to Validity

All fuzzing campaigns were run with identical seeds; some were provided by the software developers or original fuzzing harness authors, and where these were not available we used a single simple generic seed that was not specialised to the SUT. The fuzzer which LeakFuzzer is based on, AFL++, requires at least one seed to start; hence we used a fixed seed to simulate starting from an empty state as well as we could. It is possible that using the developer provided seeds may have provided an advantage that is not replicable by CBMC, as it must start from an empty state. It is also possible that different hardware setups may favour the kinds of processing that CBMC does over fuzzing (for example having more advanced vector instructions available may help), though this was controlled for on our setup by ensuring that all tests were ran on a single core. Additionally, all of the evaluation has been done with programs that contain known information leak bugs, and we have not tried to discover new, previously unknown ones. The set of pro-

grams on which we did evaluate is limited in size and scope; unfortunately such bugs are hard to find documented, and constructing harnesses for the large multi-user programs that are capable of suffering is an extremely time consuming process.

4.6 Conclusions and Future Work

We have presented LeakFuzzer, a fuzzer-based hypertesting approach to detecting leakage of confidential information, and proven its ability to do so on a varied range of benchmarks including seven information leakage related CVEs. It has outperformed a combination of state of the art existing insecure flow detecting techniques, and demonstrated its ability to scale to real-world systems of considerable size. It inherits the advantages of the type fuzzers on which it is based, most notably scalability, coverage and automated input generation. Because it stores more data than a conventional fuzzer, it uses more memory though this was not a problem in our evaluation experiments.

In the future, we hope to extend LeakFuzzer to handle programs written in other programming languages, where sanitizers are not available. Additionally we plan to quantify the flows to allow for application to a broader range of programs, such as password checkers, that need to reveal some amount of confidential information in order to function usefully.

Chapter 5

Quantifying Information Leakage With a Fuzzer – NIFuzz

5.1 Introduction

Information flow control (IFC) describes the controlled flow of information in systems. Typically, a failure of IFC that results in *secret* information being revealed to users that lack sufficient privilege is referred to as an *information leak*; such errors unfortunately go largely underreported. One paper [28] — which presents a technique for generating exploits from information leak vulnerabilities in OS kernels — states that these "vulnerabilities are usually assigned low CVSS [Common Vulnerability Scoring System] scores or [are left] without any CVEs [Common Vulnerability Enumerations] assigned".

To give an example of where IFC is required, let's consider the context of a

Linux-based OS where there is a file `secrets.txt` for which only the user 'root' has read privilege. In this case, the OS is responsible for enforcing the IFC *policy* that during OS execution information should not flow from `secrets.txt` to any user other than 'root'.

A shibboleth for IFC is some type of non-interference policy [46]. For decades now this binary classification of flow policies (flow or non-flow) have been regarded as overly restrictive, i.e. small leaks may be tolerable depending on context, such as in programs like password checkers [131]. When the leak is not tolerable, automated repair may require the size of the leak as part of a fitness function [85]. *Quantified information flow* (QIF), as the name suggests, aims to quantify the amount of information flowing from one point to another, using information theory. Typically it is used to quantify the amount of *secret* information that is allowed to flow from privileged program input to unprivileged program output. Taking the above example, if we were to execute the `read` system call on `secrets.txt` as some user other than 'root' and received the output "`P@ssword1`", we would say that the quantity of information is 9 bytes [1]. Here there is at least one unprivileged program input—the file descriptor for `secrets.txt`—and also one privileged program input; that is the contents of the file itself. The privileged program input is provided by the kernel in response to the system call requesting the file contents, and this process is opaque to the unprivileged user.

Often, these *information leaks* reveal only a fraction of the *secret*; this is true

[1] assuming each character is encoded by 1 byte

for the OpenSSL Heartbleed bug which could leak up to $\approx$ 64KiB per request, regardless of the quantity of *secret* data stored in the system. The amount of information leaked can be dependent on both the privileged and unprivileged (*secret*) inputs; for example, Heartbleed could leak between 0 and $\approx$ 64KiB depending on the unprivileged input – which was the contents of the heartbeat request packet. The maximum amount of information that can be leaked is referred to as the *channel capacity* of the leak.

In this chapter, we present a new fuzzer NIFuzz that can not only detect failures of IFC, but also provide a QIF estimate for them. Where prior approaches to QIF estimation have scaling constraints [59, 99], or do not provide automated program exploration [29, 30] — both of which make automated analysis of large-scale software systems impossible — our approach offers the scalability of fuzzing. NIFuzz provides both a strict lower-bound for channel capacity, as well as a state-of-the-art estimate of channel capacity for large leaks. Additionally we provide the first implementation of a testing-based approach that can provide a measure of conditional mutual information between program inputs and outputs. That is, the amount of leakage from *secret* privileged inputs to unprivileged outputs, conditioned on the unprivileged inputs. Conditional mutual information is a measure that is parameterised on the input distribution of the program, and gives the *expected* amount of information leaked per execution; whereas the channel capacity gives the *worst case* leakage. NIFuzz expands on concepts first introduced in LeakFuzzer, and extends these to estimate QIF rather than merely

detecting the presence of information leaks.

NIFuzz is capable of determining whether the leakage has come from an *explicit secret input*, as would be the case for the file contents example above, or *stack* or *heap* memory, as was the case for Heartbleed. To the best of our knowledge, no fully-automated, scalable tool or approach currently exists that can monitor all three sources; let alone provide feedback on which source the *secret* information came from. The combination of the quantity of information leakage and its source can be extremely useful in locating and fixing IFC related bugs. Larger leaks are likely to have more severe consequences, so awareness of the quantity of information leaked is useful for deciding on the priority of the bug. Additionally QIF could be useful in confirming that programs which are intended to leak a certain amount of information, actually do leak that amount. Finally, whilst our tool does not target side-channel leaks such as rowhammer [63], spectre [69] or meltdown [73]; the QIF estimation approaches introduced in this paper could be applied to quantifying their channel capacities which contributes to their severity.

We evaluate NIFuzz by generating QIF estimates for 14 programs containing information leaks, including 8 known CVEs, the largest program being 278k lines of code and containing leaks of up to 500,000+ physical bits. On this corpus, NIFuzz is able to identify all information leaks in all programs, and provide sensible lower-bounds and estimates for channel capacity in each case.

To summarise, the contributions of this chapter are as follows:

- Description of methodology for computing 3 different quantified information flow estimation metrics, including a method for quickly estimating the channel capacity of large leaks and a new derivation for the computation of conditional mutual information.
- Description of an approach for distinguishing the source of information leaks from explicit secret input, uninitialised stack- and uninitialised heap-memory.
- Description of an explore-exploit type mutation schedule to improve chances of discovering information leaks, and improve quantified information flow estimates.
- A publicly-available implementation of NIFuzz including all of the above, and a evaluation of its leak detection and quantification estimating abilities.

5.2 Related Work

The most closely related works are implementations of information flow control and quantified information flow; we have grouped these into four categories. Firstly, static analysis approaches, then dynamic analysis approaches distinct from fuzzing. Then fuzzing applied to side-channel leakage, and finally fuzzing applied to input-output leakage.

5.2.1 Static Analysis

An early technique capable of quantifying information flow in realistic programs is described in a 2007 paper by Clark, Hunt and Malacaria [32]. This paper uses a minimalist imperative language with static typing rules to illustrate the concepts, though no automated compiler was created and all of the example programs were manually checked. A practical model-checking based approach to QIF was proposed in a 2010 paper by Heusser and Malacaria [59], this tool was evaluated 6 fragments of real-world programs containing information leak bugs and could provide lower-bounds on channel capacity up to 7 bits. A 2014 paper by Phan and Malacaria [99] provided an alternative model checking algorithm; this was able to produce the same bounds but in considerably less time. As model checking techniques, the prior two works have scalability limitations which were overcame in a 2016 paper by Biondi et al. [14]; this paper proposed a 'hybrid' approach to QIF combining a precise program analysis with statistical methods in segments where precise analysis is too computationally complex. It was evaluated on a set of 6 constructed programs, written in a simple programming language developed for the paper, ranging in size from 10 to 33 lines of code; and hence cannot be considered to be a 'real world' approach in its describe form.

5.2.2 Dynamic Analysis

LeakiEst is a statistical tool for estimating QIF from a set of samples proposed in 2013; this operates on a simple text file containing pairs of (*secret*

input, *low output*) and is thus language agnostic in contrast to the tools above. The paper demonstrates that this is able to detect timing-channel leakage when provided the program runtime as output. MutaFlow is a lightweight mutation-based analysis [81] that mutates secret inputs at their source and observes whether a change has occurred at the program outputs. It specifically targets android in the paper's evaluation, and achieves similar results to a static analysis tool FlowDroid [8] with much lower implementation cost (requiring only 10% as many the source code lines).

5.2.3 Fuzzing Applied to Side-Channel Leakage

There are four papers introducing fuzzing-based tools to detect side-channel leakage, three of which target Java programs: JVM Fuzzing for JIT-Induced Side-Channel Detection [22], DifFuzz [94] and QFuzz [95]. DifFuzz does not attempt to quantify the leakage, only verify its presence, while the other two do provide estimates on leakage amount. Both DifFuzz and QFuzz use a count of Java bytecode instructions to approximate program runtime, as opposed to using real execution time. The fourth tool ct-fuzz [58] also detects side-channel leaks, but in C and C++ programs; to do so it treats any path divergence due to secret input as leakage, and uses a model of CPU cache to approximate memory access delays; it counts the maximum number of possible paths for any public input, so can provide some notion of a lower-bound on the channel capacity.

5.3 Quantifying Information Flow

5.3.1 Background

In this chapter we measure the quantity of information flows with two measures; both of which are measures of Shannon information, and give a quantity in *bits*. The first of which measures the maximum amount of information that can be learnt from *high* input from the *low* output in a single execution, we refer to this as *channel capacity* throughout the chapter. The second is the *conditional mutual information*; that is, the mutual information between *high* input and *low* output, conditioned on the *low* input.

In order to understand the *channel capacity* measure, let's consider the following example, with four possible outputs $0, 1, 2, 3$ visible to *low*:

```
1 uint8 target_func(uint8 low) {
2   uint8 high = getHighInput(); // read input from High
3   if (low % 4 == 0)
4     return high % 4;
5   else
6     return low % 4;
7 }
```

We can calculate the *channel capacity* described above by finding the value of *low* input that allows for the most distinctions on *low* output. Given that this is a deterministic program, any differences in *low* output must be caused by *high* input (when the *low* input is fixed). A value of *low* input for which there are most distinctions on output is 0 (or any multiple of 4); this causes the return value to come from line 4, and this return value could be $0, 1, 2$ or 3. We typically quantify the leakage in bits, which for the *channel capacity*

can be calculated by taking log_2 of the number of distinct low outputs; here that is $\log_2 4 = 2$ bits.

To calculate the *conditional mutual information* measure, we can use the following formula [35]:

$$\begin{aligned} I(X;Y \mid Z) &= H(X \mid Z) - H(X \mid Y, Z) \\ &= E_{P(x,y,z)} \log_2 \frac{P(X,Y \mid Z)}{P(X \mid Z)P(Y \mid Z)} \end{aligned} \tag{5.1}$$

Where entropy of X, denoted $H(X)$, is defined as:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i) \tag{5.2}$$

And the conditional entropy of X given Z, denoted $H(X \mid Z)$, is defined as:

$$H(X|Z) = -\sum_{z \in Z} P(z) \sum_{x \in X} P(x|z) \log_2 P(x|z) \tag{5.3}$$

Mapping *low* output to X, *high* input to Y and *low* input to Z in Equation 5.1 gives us the mutual information between *high* input and *low* output conditioned on *low* input, which works out as 0.5 bits for the above listing and secure flow policy. Without conditioning, the mutual information between *high* input and *low* output is only ≈ 0.0013 bits.

There are many other ways to quantify information flow discussed in aca-

demic literature. One uses the mutual information between *high* input and *low* output (without conditioning on the *low* input) [83]. Another paper defines a *conditional guessing entropy*, which describes the *low* attacker's uncertainty of the *high* input after making a guess and observing the result.

Quantifying information is hard. According to traditional information theory, more uncommon events contain more information. In this work, we assume that all events follow a uniform distribution, as we do not know the real input distribution for the programs being tested. Even with this assumption, quantifying information is still hard. The reason that our approach can scale to arbitrarily large programs is that it is a testing approach – meaning that only one input is tested at a time. This unfortunately means that the only true way to measure information flow is to execute every possible input. As this is not feasible for almost any interesting program, we must resort to estimating it instead. There are three different estimation metrics that we update throughout the fuzzing campaign: channel capacity lower-bound, direct mappings and conditional mutual information.

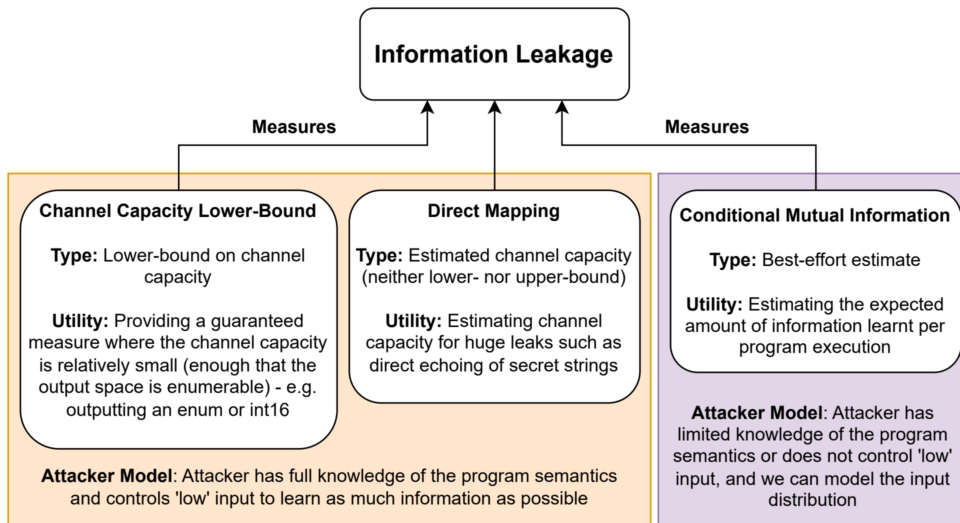


Figure 5.1: A diagram detailing the 3 approaches that we use for estimating quantifying information flow, note that the left two both measure channel capacity and all three are measures of Shannon information.

There are two attacker models that we consider. The first of which is an attacker that has full knowledge of the program semantics and can control 'low' input; the Heartbleed bug is such an example – here the attacker controls the payload of the heartbeat packet which is sent to the server. In such a case, we believe that the *channel capacity lower-bound* and *direct mapping* measures are the most useful, as it may reasonably be assumed that the attacker would choose a 'low' input that maximises the amount of information learnt.

The second attacker model is of an attacker that either lacks knowledge of the program semantics or can observe but not control 'low' input. For this attacker model, we propose the estimated *conditional mutual information* metric, which provides us with the amount of information learnt from 'secret' input, conditioned on 'low' input.

5.3.2 Estimating Conditional Mutual Information

The non-interference property guarantees that desired flow relations between the input partition and the output partition hold. Not only can we use a non-interference property as an oracle to detect violations of such policies in software but we can use information theory to rank the severity of the violations, i.e. measure the degree of interference. Let S^i be a sensitive input category in the data and O^o be an output observation category of interest while E^i ("Else") is the union of all input parts apart from S^i, so $S^i \cup E^i = In$, the random variable in the inputs. A frequentist view of probability has it that Shannon's Conditional Mutual Information quantity expresses the information shared between S^i and O^o, once one excludes any interference from the non-S^i inputs, as

$$I(S^i; O^o \mid E^i). \tag{5.4}$$

In what follows we rewrite this quantity:

$$\begin{aligned} I(S^i; O^o \mid E^i) &= H(S^i \mid E^i) - H(S^i \mid O^o, E^i) \text{ definition} \\ &= H(S^i, E^i) - H(E^i) - [H(S^i, O^o, E^i) - H(O^o, E^i)] \text{ chain rule} \\ &= H(In) - H(E^i) - [H(In, O^o) - H(O^o, E^i)] \text{ partition of In} \\ &= (H(O^o, E^i) - H(E^i)) - (H(In, O^o) - H(In)) \\ &= H(O^o \mid E^i) - H(O^o \mid In) \text{ chain rule} \end{aligned} \tag{5.5}$$

If the program is deterministic, the term $H(O^o \mid In) = 0$ as the output obser-

vations completely depend on the inputs so $I(S^i; O^o \mid E^i) = H(O^o \mid E^i)$.

When dealing with millions of inputs using a fuzzer, it is useful to minimise the amount of information that the fuzzer needs to record. Below we show that to calculate the interference in the deterministic case, a fuzzer need only record the violations of the non-interference property. We have partitioned the inputs into $S^i \cup E^i = In$. Now we further partition the *non-sensitive* input parts into those involved in violations and those not, $V \cup \overline{V} = E^i$. Define V as $V = \{e \in E^i \mid \exists s_1, s_2 \in S^i \wedge f(s_1, e) \neq f(s_2, e)\}$ where f is the semantics of a deterministic program. V induces a subset of the support for the random variable $\langle O^o, E^i \rangle$. Let us abuse notation slightly by using the name of the random variable as also the name for its support set. Define $\langle O^o, E^i \rangle_V = \{(o, e) \in \langle O^o, E^i \rangle \mid e \in V\}$ as the subset of the support set for $\langle O^o, E^i \rangle$ that has a non-sensitive input appearing in one or more failing hypertests. Assume that the program (semantics f) being fuzzed is deterministic in its behaviour.

$$
\begin{aligned}
I(S^i; O^o \mid E^i) = H(O^o \mid E^i) &= H(O^o, E^i) - H(E^i) \\
&= -\textstyle\sum_{(o,e) \in (O^o, E^i)} p(o, e) \log p(o, e) \\
&+ \textstyle\sum_{e \in E^i} p(e) \log p(e) \\
&= -\textstyle\sum_{(o,v) \in (O^o, E^i)_V} p(o, v) \log p(o, v) \\
&- \textstyle\sum_{(o,\overline{v}) \in (O^o, E^i)_{\overline{V}}} p(o, \overline{v}) \log p(o, \overline{v}) \\
&+ \textstyle\sum_{v \in V} p(v) \log p(v) + \textstyle\sum_{\overline{v} \in \overline{V}} p(\overline{v}) \log p(\overline{v}).
\end{aligned}
\tag{5.6}
$$

Since the program is deterministic and $\bar{v}$ is not part of a violation, we have the following: $\forall \bar{v} \in \overline{V}, \exists o \in O^o, \forall s \in S^i, f(s, \bar{v}) = o$. That is, if we pick a value for the non sensitive part of the input and that part is never involved in a violation, then as the secret part varies, the semantics will always produce the same output observation. So $p(\overline{v}) = p(o, \overline{v})$, since the probability distribution for E^i is a marginal distribution of that for $\langle S_i, E^i \rangle$ and o is fixed. Then we have

$$- \sum_{(o,\overline{v}) \in (O^o, E^i)_{\overline{V}}} p(o, \overline{v}) \log p(o, \overline{v}) + \sum_{\overline{v} \in \overline{V}} p(\overline{v}) \log p(\overline{v}) = 0 \tag{5.7}$$

and

$$I(S^i; O^o \mid E^i) = - \sum_{(o,v) \in (O^o, E^i)_V} p(o, v) \log p(o, v) + \sum_{v \in V} p(v) \log p(v) \tag{5.8}$$

Using this transformation, we can calculate the mutual information between secret input and low output, conditioned on low input, while keeping track *only* of the hypertests that are violations of the security policy. For programs where only a fraction of low inputs participate in a violation, this vastly reduces the amount of input-output mapping data that needs to be stored by the fuzzer and can make our estimated flow quantity more accurate.

5.3.3 Implicit and Explicit Flows

There are two types of input-output information flow; implicit and explicit. Explicit flows are somewhat obvious, but implicit flows can be less intuitive;

they are explained here to illustrate to ensure that the reader is aware of the different mechanisms by which information can flow.

In explicit flows, sensitive information is copied directly at some point, possibly to the program output by some form of assignment; for example:

```
1 int isEven(int input) {
2   return input % 2;
3 }
```

Here we treat the return value as 'output', the `% 2` operation reduces the amount of information learnt to just 1 bit (compared to the 32 or 64-bit `input`), but still explicitly returns a value directly derived from `input`. This explicit flow can pass through any number of variable assignments or operations:

```
1 int fifthBitNotSet(int input) {
2   int shifted = input >> 5;
3   int masked = shifted & 1;
4   int isSet = masked;
5   return !isSet;
6 }
```

This time we perform a more complicated set of operations, but information from `input` still is explicitly passed through to the return value.

In implicit flows, information is learnt from the control flow of the program, for example:

```
1 const char *isEven(int input) {
2   if (input % 2 == 1)
3     return "false";
4   else
5     return "true";
6 }
```

In the above program, there is no direct *explicit* assignment from `input` to the return value; however receiving the return value `"false"` *implies* that

the conditional check on line 2 evaluates to true, which in turn means that the value of `input % 2` is 1.

All three of the above explicit and implicit flow examples leak exactly 1 bit of information, just through different means.

5.3.4 Direct Mappings Between Input and Output

> **Note:** from this point onwards, we substitute the terms *high* and *low* that are typical in the theoretical IFC literature, with *secret* and *public*; where a typical security policy allows information to flow from *public* to *secret* but not vice versa.

In order to calculate the channel capacity lower-bound by counting the number of unique public outputs for a given public input, $2^{\texttt{bits_leaked}}$ outputs must be found. For cases where the number of bits leaked is high, a true calculation of this quantity through this approach becomes infeasible – to quantify 30 bits, > 1 billion outputs must be found. However, in some programs, e.g. HeartBleed and other leaks from memory regions, the semantics of the vulnerability means that there is effectively a one to one mapping between the secret and the observations – and we can exploit this directly. If we can observe that flipping a bit in the input results in a bit in the output being flipped, we can approximate the size of these large leaks.

We propose building a mapping from *secret* input bits to corresponding *public* output bits. We say that a mapping exists between a *secret* input bit and a *public* output bit if flipping the given input bit (either from 0 to 1 or vice

versa), also results in the output bit being flipped. The simplest trivial example of this is as follows, again we treat return value as output:

```
int identity(int input) {
  return input;
}
```

If we consider fixed precision `ints` in their binary form, we can see that flipping any bit of `input` results in the same bit of the output being flipped. For example, the input `0b00` trivial produces the output `0b00`. Flipping the first bit of the input gives us `0b01`, and the corresponding output `0b01`. Instead, flipping the second bit of the input gives us `0b10` and the output `0b10`. The mapping here can be extended to the full set of bits that make up `input`; if using 32-bit `ints`, this would require discovering 2^{32} ($\approx$4 billion) unique outputs.

As a trivial example, the above could be done by searching for exact matches between input and output bits. To demonstrate the advantage of bitflips, let's consider a slightly trickier program:

```
int bitwiseNotted(int input) {
  return ~input;
}
```

Here we apply a bitwise NOT to the return value; the input `0b00` produces output `0b11...11` (all 32-bits are now set to 1) and input `0b01` produces output `0b11...10`. If we were to simply search for occurrences of the provided input in the corresponding output, we would find none; but observing the position of flipped output bits still works.

This concept to allow fast approximations of quantities for large leaks is ex-

panded upon in section 5.6.3 with the description of an algorithm to produce such a mapping automatically.

5.4 Technique and its Implementation

NIFuzz has been implemented in Rust using components from LibAFL [3], and the source code is publicly available[2]. Where there is conceptual overlap with LeakFuzzer from the previous chapter, this has been reimplemented from scratch in order to take advantage of the higher-level programming constructs available in Rust (as opposed to C). In some cases this reimplementation has benefited from the lessons learnt in the development and evaluation of LeakFuzzer. We have chosen to implement an input-output leakage measurement, however the technique could be applied to side-channels such as timing or power with an appropriate channel quantification methodology. In our input-output implementation we use a forkserver to run the SUT (system under test) and inherit the `stdout` and `stderr`; this allows for a higher throughput than running the SUT as a command (via exec or similar). Additionally we use the standard LibAFL coverage map feedback to guide the search, this stores any inputs that discover new branches to a corpus for further mutation.

The following sections discuss the specifics of NIFuzz that diverge from a typical AFL-like fuzzer. Firstly we discuss *input structure* which is needed for the fuzzer to distinguish whether output changes are due to *secret* input

[2] `https://github.com/DanBlackwell/NIFuzz`

as opposed to *public*. Then we look at the *mutation phase*, which is split into *explore and exploit* subphases. This is followed by the specifics of how NIFuzz maximises the observable quantity of leakage using specific mutations. Finally we discuss how QIF estimates are produced from the data stored by the fuzzer.

5.5 Input Structure

In order for the fuzzer to know that it has discovered a leak, it must have found two inputs with matching *public* parts (and differing *secret* parts) that produce differing outputs (assuming that the program is deterministic); therefore the fuzzer must be aware of which parts of the input are *secret*. Hence input structure is a key ingredient.

5.5.1 Explicit Inputs, Stack Memory and Heap Memory Secrets

NIFuzz has the ability to determine whether leaked information has come from explicit *secret* input, uninitialised stack memory or uninitialised heap memory. This is beneficial as once a leak has been discovered, this information can be used to help pinpoint the program point at which the *secret* information started being treated as *public* information.

To make this possible, both the internal and external representation of program inputs have 1 *public* and 3 distinct *secret* input parts: explicit input,

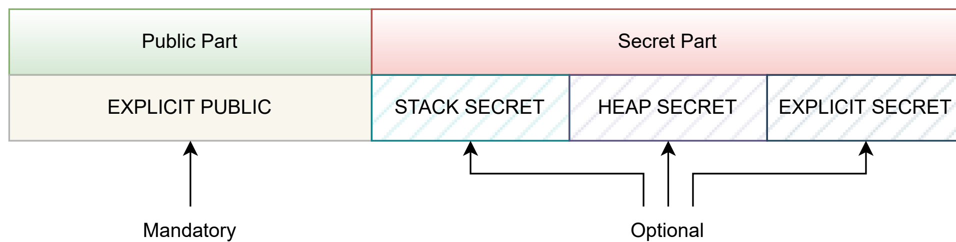


Figure 5.2: Diagram showing the structure of an input, as seen from NIFuzz point of view. Note that each subsection of the secret part of the input is optional.

stack memory and heap memory. We provide an easy method for retrieving each of these parts within the fuzzing harness, and function implementations to populate the uninitialised portions of stack[3] and heap memory with the provided values.

Additionally, it is possible for programs to only use the parts of the input that are necessary, for example some may not use explicit secret input. NIFuzz will not populate these parts of the input nor waste time attempting to mutate them. Note that whilst it would be possible to run NIFuzz with no secret parts at all, it would be better to use a standard grey-box fuzzer instead, as the mutation engine and corpus schedulers are optimised towards detecting information leaks.

To illustrate an example of stack memory leakage, the following code is taken from the Linux kernel (CVE-2009-2847) as used in our evaluation:

[3]for x86 and AArch64

```
1  typedef struct sigaltstack {
2    void __user *ss_sp;
3    int ss_flags;
4    size_t ss_size;
5  } stack_t;
6
7  int
8  do_sigaltstack (const stack_t __user *uss,
9                  stack_t __user *uoss,
10                 unsigned long sp)
11 {
12   stack_t oss;
13   ...
14   if (uoss) {
15     oss.ss_sp = (void __user *) current->sas_ss_sp;
16     oss.ss_size = current->sas_ss_size;
17     oss.ss_flags = sas_ss_flags(sp);
18   }
19   ...
20   if (uoss) {
21     if (copy_to_user(uoss, &oss, sizeof(oss)))
22   ...
23   }
24 }
```

In the Linux kernel, variables marked with `__user` (such as `ss_sp` on line 2) come from untrusted sources; these are treated as *public* input in our evaluation. The function `copy_to_user` on line 21 copies the memory contents of the kernel-space variable `oss` to the user-space `uoss`. We can see that `oss` is declared on line 12, but it is not initialised; however on lines 15–17 all three struct member values are populated. A reasonable assumption would be that besides those three struct members, no further information is disclosed from kernel- to user-space. In fact, there are 4-bytes of potentially sensitive kernel-space memory leaked due to struct *padding*. The way that the `sigaltstack` (aka `stack_t`) struct is laid out in memory is shown in the following diagram:

Mem. Address	0-3	4-7	8-11	12-15	16-19	20-23
Variable	ss_sp		ss_flags		ss_size	

On 64-bit architectures, the compiler prefers to align variables to 8-byte (64-bit) boundaries, indicated here by double vertical lines, this is done for performance reasons. Additionally, it must lay out struct members in the order that they are defined. As `ss_flags` is a 4-byte integer, the compiler inserts 4 bytes of padding after it, so as to align `ss_size` with a boundary. So despite setting values for all struct members, bytes 12-15 in the above diagram are never initialised, and therefore still hold the value of the variable previously stored in this area of memory. Thus, when `copy_to_user` is called, these 4-bytes — containing potentially sensitive kernel-space information — are copied out to user-space in the `uoss` variable. A simple fix in this case is to add a call to `memset` the memory containing `oss` to 0 before populating the struct member values.

In this particular example, the memory containing the leaked information was on the stack, as `oss` is a local function variable. If instead `oss` was allocated on the heap with `malloc`, then it would have been heap memory contents leaked instead. By detecting that the leak is 4-bytes, comes from the stack, and from a leak within this function, it should be possible to find the source of the bug and produce a patch.

To illustrate an example of explicit *secret* input leakage, we use the following code taken from Fedora's NetworkManager (CVE-2011-1943) as used in our evaluation:

```
1  void
2  nm_setting_vpn_add_secret (NMSettingVPN *setting,
3                             const char *key,
4                             const char *secret)
5  {
6    ...
7    g_hash_table_insert (
8      NM_SETTING_VPN_GET_PRIVATE (setting)->secrets,
9      g_strdup (key), g_strdup (secret));
10 }
11
12 static void
13 destroy_one_secret (gpointer data)
14 {
15   char *secret = (char *) data;
16
17   /* Don't leave the secret lying around in memory */
18 g_message ("%s:␣destroying␣%s", __func__, secret);
19   memset (secret, 0, strlen (secret));
20   g_free (secret);
21 }
```

Here the `secret` parameter to the function `nm_setting_vpn_add_secret` is an explicit secret input; it is expected by the user that this secret should only be used within the NetworkManager system. As seen on line 18, the secret is printed in plain-text by a logging function. We would assume, due to the incorrect indentation and preceding comment, that this particular line of code was added for debugging and mistakenly made its way into release builds.

5.6 Mutation Phases

Now that we have covered the structure of the inputs, we can explain the mutation phase. The mutation phase is split into an *explore* stage that aims

to detect *violations* of the security policy (i.e. information leaks), and *exploit* stage which aims to quantify these leaks. The *explore* stage is run exclusively until a violation is found, and then run interchangeably with the *exploit* stage afterwards.

5.6.1 Explore Stage (leak discovery)

As hinted at above, the *explore* stage is the default mode of NIFuzz. It aims to explore program functionality using the typical coverage-guided grey-box fuzzing methodology, but also search for the presence of a violation (that is, a *public* input that produces different program *public* output depending on the value of *secret* input). When we refer to *public output*, we are referring to output which is visible to *public* level users. The following steps are repeated:

1. Select an input from the fuzzer input corpus.
2. Mutate the *public* part of the input, execute the target program with this input and collect the output.
3. Mutate the *secret* parts of the input, execute the target program with this input and collect the output.
4. Mutate the *public* and *secret* parts of the input, execute the target program with this input and collect the output.

Both the *public* and *explicit secret* (if present) parts of the input independently have traditional fuzzing mutations applied, such as bitflips or splicing. The *stack memory secret* and *heap memory secret* parts of the input have a simple

mutation applied, in (3) they are set to `0b10101010` and `0b01010101`. The functions used to fill uninitialised memory in the SUT repeat the relevant part of the input until the memory area is completely filled. These two 1-byte inputs therefore fill the uninitialised memory with opposing bit values resulting in any direct disclosure from uninitialised memory producing a different observable output.

Note that we do not aggregate (2) and (3), as in a conventional fuzzer, since the program may be sensitive to secret inputs and also mutating public parts alone may be necessary to reach new coverage. Imagine an input for the following program with `{ secret: 1, public: 0 }`:

```
1 if (secret == 1) {
2   if (public < 3) return 0;
3   else return 1;
4 } else {
5   return 0;
6 }
```

We could only reach the *else* branch on line 3 by retaining the current value of `secret` and mutating `public` to be >= 3.

If a pair of tests with matching *public* inputs, but differing *public* outputs (due to *secret* input rather than non-determinism), are found then these are stored and the *public* input is added to the set of *violations*.

We cache information about every input and output along the way in order to maximise our odds of detecting a violation. The fuzzer state maintains a `map` as shown in figure 5.3 (note that this object is simplified, and in our implementation, hashes are stored rather than the full byte array wherever

```
type OutputData {
  stdout: [byte],
  stderr: [byte]
}

type SecretInputParts {
  // the '?' here indicates an optional value, one or more
  // members must be populated
  explicit_secret: [byte]?,
  stack_memory_secret: [byte]?,
  heap_memory_secret: [byte]?
}

// The IOHashValue object is defined as follows
type IOHashValue {
  // each unique public output is stored with a corresponding
  // secret input
  secret_input_for_public_output:
      HashMap<OutputData, SecretInputParts>,
}

// `PublicInput` is a type alias for list of `byte`
typedef [byte] PublicInput;

// This stores a mapping from 'public input' to an
// `IOHashValue` object
map: hashmap<PublicInput, IOHashValue>
```

Figure 5.3: Simplified description of the `map` stored in the fuzzer state

possible).

5.6.2 Exploit Stage (leak quantification)

This stage operates on a violation, thus will not run until a violation has been discovered. At a high-level, the stage operates as follows:

1. Select an input from the set of violations.

2. See whether we have checked for direct mappings from secret input

bits to output bits (i.e. flipping a particular bit from the *secret* input results in one or more output bits flipping).

3. If no, then perform the set of checks to determine whether such mappings exist (see section 5.6.3).

4. If yes, then sample secret inputs from uniform and execute them in order to collect the output.

5.6.3 Bitflips

This is the first phase of mutations applied during the *exploit* phase on a newly discovered violation. The aim is to produce a direct map from secret input bits to public output bits; that is a mapping whereby flipping this bit in the input results in the one (or more) bit in the output being flipped. Not all information leaks will have such a mapping. The search procedure is as follows:

For each secret part of the input present (from the possibilities *explicit secret input*, *stack memory secret* and *heap memory secret*), flip all bits of this part and run the SUT with this input. If the output is changed, then store all bits of this part to the list *influences*. Note that each member of *influences* is essentially a tuple made up of (secret_input_part, bit_number).

Then, if the number of elements in *influences* is < 1000, we flip one bit at a time in order to find potential mappings from secret input to output bits (as described in algorithm 1). Otherwise there are more than 1000 potential

bits to flip, so we use a faster approximation that needs $\lceil \log_2 |influences| \rceil$ steps to estimate all mappings (algorithm 2). Figure 5.4 shows a worked example with a small program leaking bits 3 and 6 from secret input directly to output.

Algorithm 1: Simple algorithm for flipping 1 bit at a time in order to find potential mappings from secret input to output bits.

Input: `SUT` - The program being tested
Input: `input` - An array of bits making up the original program input
Input: `influences` - The set of bit positions in the input that map directly to output bits
Output: `bitflipMap` - A hashmap from secret input bit positions to public output bit positions

```
originalOutput ← executeWithInput(SUT, input)
seenBitflips ← ϕ                              // Empty Set
bitflipMap ← {}                           // Empty HashMap
for i in influences do
    mutatedInput ← input
    mutatedInput[i] ← mutatedInput[i] XOR 1
    output ← executeWithInput(SUT, mutatedInput)
    outputFlips ← findFlippedBits(originalOutput,
     output)
    manyToOnes ← seenBitflips ∩ outputFlips
    for (_, outputFlips) in bitflipMap do
        outputFlips ← outputFlips \ manyToOnes
    end
    seenBitflips ← seenBitflips ∪ outputFlips
    bitflipMap[i] ← outputFlips \ manyToOnes
end
bitflipMap ← bitflipMap.filter(lambda(k, v){|v| >
 0})
```

Algorithm 2: Fast algorithm for finding potential mappings from secret input bits to output bits

Input: `SUT` - The program being tested
Input: `input` - An array of bits making up the original program input
Input: `influences` - The set of bit positions in the input that map directly to output bits
Output: `bitflipMap` - A hashmap from secret input bit positions to public output bit positions

```
originalOutput ← executeWithInput(SUT, input)
outputToInputMap ← {:}            // Empty HashMap
for i ← 0 to originalOutput.len do
    outputToInputMap[i] ← 0
end
for bit ← 1 to ⌈log2(influences.len)⌉ do
    bitValue ← 1 « (bit - 1)
    mutatedInput ← input
    for index ← 0 to input.length - 1 do
        if (index / bitValue) % 2 == 1 then
            input[i] ← input[i] XOR 1
        end
    end
    output ← executeWithInput(SUT, input)
    outputFlips ← findFlippedBits(originalOutput,
     output)
    for i in outputFlips do
        outputToInputMap[i] ← outputToInputMap[i] +
         bitValue
    end
end
bitflipMap ← {:}                  // Empty HashSet
for (outputBit, inputBit) in outputToInputMap do
    if inputBit > 0 then
        bitflipMap[inputBit] ← bitflipMap[inputBit]
         ∪ outputBit
    end
end
```

Figure 5.4: A worked example demonstrating how algorithm 2 can find the mapping between `secret` input bits 3 and 6, and the output bits 3 and 6.

```
1 if (public == 0)
2   return secret & 0b01001000
3 else
4   return 0
```

```
input = { public: 0b0, secret: 0b00000000 }
originalOutput = 0b00000000
outputToInputMap = {}
```

bit	*bitValue*	*mutatedInput*	*output*	*outputFlips*	*outputToInputMap*
1	1	0b10101010	0b00001000	[3]	{ 3: 1 }
2	2	0b11001100	0b01001000	[3, 6]	{ 3: 3, 6: 2 }
3	4	0b11110000	0b01000000	[6]	{ 3: 3, 6: 6 }

after inverting the mapping: `bitflipMap = { 3: { 3 }, 6: { 6 } }`. i.e. bits 3 and 6 of `secret` map directly to output.

The issue with both algorithms for detecting mappings is that output bits could be affected by non-determinism, resulting in spurious mappings. We observed this in Heartbleed, where a particular 16-byte block of memory would be populated with 'random' values – it is possible that these were generated using system time or `/dev/urandom` or similar. To overcome this we follow up with a series of tests with combinations of the mapped input bits flipped and check that we see the expected output. Any bits that flip in the output that were not predicted by the mapping are filtered out of any other mappings, as there is a many-to-one relationship. Any bits that were expected to flip in the output but did not are also filtered out. These checks are performed with all mappings at once, then $\frac{3}{4}$ of all mappings, all the way down to $\frac{1}{8}$ of all mappings; these are selected from uniform and the sampling continues until the set of mappings is stable (i.e. none are being

filtered).

5.6.4 Extending Secret Inputs - after bitflips

If there are any one-to-many mappings between secret input and public output bits, it is possible that the input is being copied repeatedly into the output up to some range. This is particularly prevalent in the *stack memory secret* and *heap memory secret* where our initial mutations during the explore stage set the length to just 1 byte, which is repeatedly copied into memory by the fuzzing harness. With a single byte input, there can be a maximum of one byte (8 bits) of leakage per run; extending the input allows us to quantify a greater amount of leakage. In order to find how much to extend the input by, the mappings are iterated through to find the greatest difference between two output flip positions (for the same input bit). Take for example the following program, and its corresponding map, generated using one of the prior algorithms:

```
1  // int32 is a 32-bit (4 byte) integer type
2  int32 targetFunc(int32 public) {
3    int32 x;
4    if (public == 1) {
5      return x;
6    } else {
7      return 0;
8    }
9  }
10
11 int32 main(void) {
12   int32 public = getPublicInputFromFuzzer();
13   byte[] stackSecret = getStackSecretFromFuzzer();
14   fillStack(stackSecret);
15   print(targetFunc(public)); // output the result
16   return 0;
17 }
```

After calling `fillStack` with (for example) `stackSecret = [0xAA]`, stack memory looks like: `{0xAA, 0xAA, ..., 0xAA}`. The corresponding `bitflipMap` is:

$\{0 : \{0, 8, 16, 24\}, 1 : \{1, 9, 17, 25\}, 2 : \{2, 10, 18, 26\}, 3 : \{3, 11, 19, 27\},$

$4 : \{4, 12, 20, 28\}, 5 : \{5, 13, 21, 29\}, 6 : \{6, 14, 22, 30\}, 7 : \{7, 15, 23, 31\}\}$. In order to find the length that we should extend the secret input `stackSecret` by, we iterate through each value in the mapping (a set of mapped output bits), and subtract the smallest from the largest value. For example for key 0, the value is the set $\{0, 8, 16, 24\}$, so the maximum difference is $24-0 = 24$. We do this for each element in `bitflipMap` and store the maximum difference; for this example, every element has the same difference, 24.

After finding this maximum value, we then extend the relevant part of the secret input by this length by repeatedly copying the current buffer until it is filled. For example, extending `stackSecret = [0xAA]` by 24 bits it

becomes

`[0xAA, 0xAA, 0xAA, 0xAA].`

We then rerun the bitflip mapping algorithm; which in this case produces a one-to-one mapping $\{0 : \{0\}, 1 : \{1\}, \ldots, 31 : \{31\}\}$. Now instead of having 8 bits mapped directly from `stackSecret` to output, we have 32 bits; which is the ground truth channel capacity for the program.

5.7 Uniform Sampling

After the bitflips have been completed—whether any direct mappings have been discovered or not—the fuzzer moves onto the next input. Once the violation corpus contains at least one input, the fuzzer has a random 50/50 chance of selecting the next input from either the main corpus (populated according to program coverage) or the violation corpus.

If a violation is selected for the second time, i.e. the bitflip map has already been generated, then we begin a new mutation process: uniform sampling. This is important for the conditional mutual information calculation, as it uses probabilities, which can be more accurately estimated using sampling with a known distribution. Additionally, we can use the number of distinct outputs observed during this process to provide a lower-bound on the *channel capacity*.

Populating the secret parts of the input from uniform is done by applying a single mutator that we have implemented as part of NIFuzz. This mutator

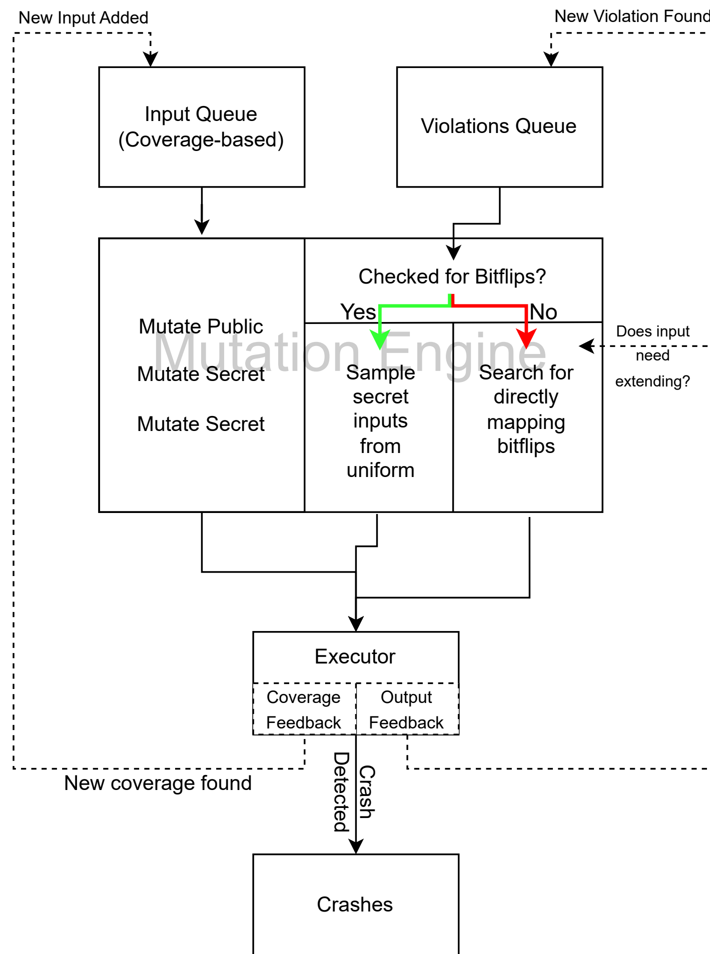


Figure 5.5: Lifecycle of an input in NIFuzz.

replaces each byte from the relevant parts of the input with values produced by a random generator. By doing this, the length of each part of the input remains unaltered, only the values change. We then execute the target program with this input and store the mapping from secret input to public output in a distinct hashmap for uniform sampled inputs (with each *violation* having its own hashmap).

In our implementation, we run this uniform sampling mutation 2^{16} (65,536) times before allowing the fuzzer to select a new input from a corpus for fuzzing.

5.8 Calculating QIF

After finishing the mutation stage on a violation, we recalculate and output the quantified information flow statistics. There are three values that we calculate:

1. estimated conditional mutual information.
2. channel capacity lower-bound; that is, the most information that can be learnt about the secret value from a single execution.
3. Estimated most bits leaked directly to output; that is, the highest number of secret input bits that map directly to public output for a single violation.

The way that these values are calculated is explained in the following subsections.

5.8.1 Estimated Conditional Mutual Information

For each public input, we store an `IOHashValue` as was shown in figure 5.3. In order to estimate conditional mutual information, we store some extra fields on the object:

```
1  // dict stores a mapping from 'public input' hash to an
2  // IOHashValue object
3  map: HashMap<uint64, IOHashValue>
4
5  type IOHashValue {
6    secret_input_for_public_output:
7          HashMap<OutputData, SecretInputParts>,
8    // hits is incremented each time this public input
9    // gets run
10   hits: uint64,
11   // the following stores a map from public output hash
12   // to a list of secret input hashes that produced
13   // this output. But only for those generated by the
14   // uniform sampling mutator
15   uniform_pub_outs_to_sec_ins: HashMap<uint64, [uint64]>,
16   // The following stores the same, but for outputs
17   // generated during non-uniform stages of fuzzing (for
18   // example during the explore stage, or bitflips)
19   non_uniform_pub_outs_to_sec_ins: HashMap<uint64, [uint64]>
20 }
```

Figure 5.6: Data structure for storing input-output mappings in fuzzer state

Using these values we can calculate an estimate with the formula derived in section 5.3.2:

$$I(S^i;O^o|E^i) = -\sum_{(o,v)\in(O^o,E^i)_V} p(o,v)\log p(o,v) + \sum_{v\in V} p(v)\log p(v) \tag{5.9}$$

We will tackle the simpler second $\sum$ first; we can get all violations V from the corpus specifically for storing *violations*. We can then calculate the probability $p(v)$ for each violation with $\frac{1}{number\ of\ unique\ public\ inputs}$. The number of unique public inputs is the number of key-value pairs in `map`. We have found that more accurate estimates can be found earlier in the fuzzing campaign by only counting those public inputs that have more than a certain number

of `hits`. This is likely to be because public inputs that have been sampled few times may be undetected *violations*.

The first $\sum$ again requires only the violations corpus from the fuzzer, however we also need the set of public output–public input (O^o, E^i) pairs. To obtain these with the most accurate probabilities, we use the `uniform_pub_outs_to_sec_ins` map for each of the violations. We sum up the length of each list for each value in the map, giving us the total number of uniform samples for this violation. We can then calculate $p(o, v)$ for each public output in the map (i.e. each key present), by multiplying the $p(v)$ value by the number of list elements and divide this by the total number of uniform samples. As some public outputs may depend on very specific secret inputs — and these may be near impossible to generate through uniform sampling, but have been found by the biased, heuristic-driven fuzzer mutation engine — we also include any public outputs found in `non_uniform_pub_outs_to_sec_ins`. As we assume these to be rare, each is given a probability of 1 $\div$ 'total number of uniform samples' for this violation. While the number of uniform samples is low, these outputs will be over-represented (i.e. be treated as having a higher probability than they really do), however as more uniform samples are collected this error is reduced.

Using the above data structure, our approach is shown in algorithm 3.

We additionally supply a command line flag that can force uniform selection of public inputs at all times; this is necessary for programs with large

variations in branch probability. Take, for example, a basic program having 2 paths with 1% and 99% probability respectively. Due to the way that the fuzzer works by selecting at random from the input corpus, it is likely that the 1% probability branch will be oversampled as whenever that input is selected from the corpus and mutated there is likely to be a much higher chance of generating a new input taking the same path.

Algorithm 3: Our approach for estimating conditional mutual information using elements of the fuzzer state shown in figure 5.6.

```
uniquePublicInputs ← 0
for (key, value) in map do
    /* Here, MIN_HITS is some constant used to
       filter out undersampled violations          */
    if value.hits >= MIN_HITS then
        uniquePublicInputs ← uniquePublicInputs + 1
    end
end
sumProbOutputs ← violations.len ÷
 uniquePublicInputs
sumProbViolations ← 0
for v in violations do
    hashVal ← map[v.publicInputHash()]
    numUniformSamples ← 0
    uniformMap ←
     hashVal.uniform_pub_outs_to_sec_ins
    for (publicOutHash, secretInHashes) in uniformMap
     do
        numUniformSamples += secretInHashes.len
    end
    for (publicOutHash, secretInHashes) in uniformMap
     do
        prob ← secretInHashes.len ÷
         numUniformSamples
        sumProbViolations += prob · log2(prob)
    end
    uniformKeys ← uniformMap.keys()
    nonUniformKeys ←
     hashVal.non_uniform_pub_outs_to_sec_ins.keys()
    uniqueNonUniforms ← nonUniformKeys \
     uniformKeys
    for publicOutHash in uniqueNonUniforms do
        prob ← 1 ÷ numUniformSamples
        sumProbViolations += prob · log2(prob)
    end
end
conditionalMutualInformation ← -sumProbOutputs +
 sumProbViolations
```

5.8.2 Channel Capacity Lower-Bound

Channel capacity is the largest quantity of information that can leak from secret input to public output in a single execution of the target program. This quantity of leakage is likely to only occur when certain inputs are provided.

NIFuzz produces a lower-bound for this particular value, as finding the exact quantity through search would require testing every public–secret input pair. Again we use the fuzzer violation corpus, this time fetching the corresponding `IOHashValue` for each violation. We then find the number of elements in the set comprised of the union of the keys in the uniform and non-uniform maps from public output hash to secret input hash. Finding the highest number of elements in any of these sets gives us the maximum number of distinctions on output (that we have discovered so far), and the $\log_2$ of this value is our lower-bound on channel capacity in bits. A pseudocode description of this is shown in algorithm 4.

5.8.3 Estimated Most Bits Leaked Directly to Output

While our channel capacity lower-bounding approach can provide a guaranteed lower-bound, it requires at least as many executions as there are distinctions on public output. For programs containing large leaks, such as the OpenSSL Heartbleed bug which leaks up to 64KiB per execution, it would take an infeasible amount of time and computer memory to attempt to produce a bound close to the true channel capacity. The direct bitflip map between secret input and public output provides a way to estimate these large

Algorithm 4: Our approach for producing a lower-bound on *channel capacity* using the elements of fuzzer state shown in figure 5.6.

```
maxOutputDistinctions ← 0
for v in violations do
    hashVal ← map[v.publicInputHash()]
    uniformKeys ←
     hashVal.uniform_pub_outs_to_sec_ins.keys()
    nonUniformKeys ←
     hashVal.non_uniform_pub_outs_to_sec_ins.keys()
    uniquePublicOutputs ← (uniformKeys ∪
     nonUniformKeys).len
    if uniquePublicOutputs > maxOutputDistinctions
     then
        maxOutputDistinctions ← uniquePublicOutputs
    end
end
```

quantities. Whilst we do a phase of testing combinations of these bitflips to search for interference between mappings, it would take the $2^{|\texttt{mappedBitflips}|}$ to exhaustively test them all; which would be the same as the number of distinctions on output if they did all map perfectly. It is therefore possible that there may be interference between mappings that were not detected, and thus the true leakage is lower than the number of bitflips in our map. It is also possible that there are more distinct outputs, for example:

```
1  if (secret & 0b111 == 0b111)
2    return 0b100
3  else
4    return secret & 0b11
```

In the above program, the input `{secret: 0}` maps to the public output 0, `{secret: 1}` maps to 1, and `{secret: 2}` maps to 2. In producing our map by flipping single bits, we would have the following map from secret

input to public output: $\{0 : \{0\}, 1 : \{1\}\}$. This would suggest a leak of 2 bits, or 4 distinctions on output; however there is a 5th when the input `{secret: 7}` is given, where the output is 4.

Despite being an oversimplification, and capable of under- or over-approximation, the directly mapped bitflips can allow for a better estimation of leakage than the other estimators for very large leaks.

The bitflip map is stored in the `IOHashValue`, so finding the estimated most bits leaked directly is simple; iterate through the violations corpus and find the number of elements in the largest map.

5.9 Evaluation

In this evaluation we aim to answer the following research questions:

RQ1: How does the lower-bound on channel capacity computed by NIFuzz compare to the ground truth channel capacity for known information leaks?

RQ2: How does the estimate on channel capacity through direct input-output bit mapping compare to the ground truth channel capacity for known information leaks?

RQ3: How does the estimate on conditional mutual information perform on a program with known input distribution?

In order to answer these questions, NIFuzz has been evaluated on a range of SUTs (Systems Under Test) including 8 information leak CVEs. Channel

SUT name	CVE number	Source	LoC	Ground Truth Channel Capacity Bits		
				Explicit Input	*Stack Mem.*	*Heap Mem.*
AppleTalk	CVE-2009-3002	[59]	149	-	80	0
cpuset	CVE-2007-2875	[59]	80	-	0	2^{31}
sigaltstack	CVE-2009-2847	[59]	110	-	32	0
tcf_fill_node	CVE-2009-3612	[59]	788	-	16	0
getsockopt	CVE-2011-1078	[99]	142	-	8	0
Heartbleed	CVE-2014-0160	[49]	278,907	-	0	>500,000
NetworkManager	CVE-2011-1943	[16]	115,643	320	0	0
RDS	CVE-2019-16714	[16]	69,163	-	16	0
IFSpec_Banking	-	[57]	97	1	0	0
IFSpec_Password	-	[57]	66	16	0	0
IFSpec_Reviewers	-	[57]	105	~18.5	0	0
expl_sec_701_bit	-	New	34	701	0	0
heap_4808_bit	-	New	34	-	0	4,808
stack_17768_bit	-	New	33	-	17,768	0

Figure 5.7: SUT name is chosen arbitrarily based on the functionality of the tested code (explicit_secret_701_bit has been shortened to expl_sec_701_bit). Lines of code was calculated using cloc, and counts the number of 'C', 'C++' and 'C/C++ header' code lines. IFSpec_* benchmarks were translated manually from Java to C. All leakage is for code compiled with gcc targeting x86_64 CPU, otherwise leakage from struct padding bytes may vary.

capacity varies between 1 bit and in excess of 400,000 bits, and leaks come from explicit secret input, uninitialised stack memory and uninitialised heap memory.

As NIFuzz is a novel concept capable of scaling to program and leak sizes far beyond previous approaches for quantifying leaks in real-world C programs [99, 59], but does not claim to have the guarantees of formal verification, it has been evaluated independently.

Testing Environment

All tests were run inside Docker containers based on the Ubuntu 20.04 distribution of Linux. Fuzzing experiments were run with 10 fuzzing campaigns in parallel, each bound to a single CPU core. Tests were run on a dedicated

server equipped with 2 x Intel Xeon E5-2620 v2 processors, making for a total of 12 cores (24 threads) at 2.10GHz, and 128GB RAM. Each benchmark was fuzzed for 12 hours. Each experimental setup/run was repeated 10 times. The nondeterministic nature of the mutation engine results can differ greatly over 12 hours.

5.9.1 Experimental Results

A summary of results are shown in figure 5.8.

RQ1: How does the lower-bound on channel capacity computed by NIFuzz compare to the ground truth channel capacity for known information leaks?

For the programs with relatively small leaks ($<$ 20 bits) – `tcf_fill_node` (16 bits), `getsockopt` (8 bits), `RDS` (16 bits), `IFSpec_Banking` (1 bit), `IFSpec_Password` (16 bits) and `IFSpec_Reviewers` ($\approx$ 18.5 bits) – the median lower-bound found in the runs is generally close to the ground truth. Due to the relatively few number of outputs needing enumeration, the ground truth was discovered for `getsockopt` and `IFSpec_Banking` in all runs. In `RDS`, all runs came extremely close, finding at least 15.96 out of the 16 bits with the closest finding 15.9999 bits.

The largest discrepancy amongst the small leaks was `IFSpec_Password`, which is due to the fuzzing harness. The *public* input in this program is a series of password guesses; in order to separate the single string provided by the fuzzer into substrings, we split on `\n` characters. The output observed is proportional to the number of guesses, with the program outputting “"No

more password tries allowed" for each guess after the *secret* value `maxTries` has been reached. As a result, in order to reach the true channel capacity, a *public* input containing >65,536 `\n` characters would need to be generated; which is very unlikely unfortunately.

The highest amount measured in any run is 19.5 bits in one run on `NetworkManager`; in that run there was a single violation for which 738,645 distinct outputs were found (by mutating the *secret* part of the input). This is still far below the ground truth lower-bound of 320 bits, and thus serves as good motivation for the coarser direct mapping approach for estimating channel capacity.

RQ2: How does the estimate on channel capacity through direct input-output bit mapping compare to the ground truth channel capacity for known information leaks?

We see that for 10 of the 14 programs, the estimate on channel capacity is much higher than the lower-bound discussed previously. For one of the other 4 programs, `getsockopt`, both estimators produced the ground truth value.

For the three constructed programs—`expl_sec_701_bit`, `heap_4808_bit` and `stack_17768_bit`—every run found the ground truth values for channel capacity. These act as a sanity check to suggest that there is no over- or under-estimation from the algorithm in ideal circumstances, even when dealing with huge leaks.

For 5 of the 8 real-world CVEs, the median channel capacity discovered

SUT	Direct Mapped Bitflips *Explicit Secret*			*Stack Memory*			*Heap Memory*			Channel Capacity (Bits)		
AppleTalk	0	0	0	80	80	80	0	0	0	16.3	16.5	17.1
cpuset	0	0	0	0	0	0	8	128	128	8	13.9	18.8
sigaltstack	0	0	0	32	32	32	0	0	0	16.6	17.7	18.9
tcf_fill_node	0	0	0	0	0	0	16	16	16	11.9	15.3	15.7
getsockopt	0	0	0	8	8	8	0	0	0	8	8	8
Heartbleed	0	0	0	0	0	0	0	443,960	517,440	0	16	17.7
NetworkManager	107	121.5	135	0	0	0	0	0	0	17.9	18.5	19.5
RDS	0	0	0	16	16	16	0	0	0	15.96	15.999	15.9999
IFSpec_Banking	0	0	1	0	0	0	0	0	0	1	1	1
IFSpec_Password	1	1	1	0	0	0	0	0	0	4.75	5.02	5.43
IFSpec_Reviewers	6	7.5	19	0	0	0	0	0	0	14.8	16.1	17.2
expl_sec_701_bit	701	701	701	0	0	0	0	0	0	15.8	16.8	17.5
heap_4808_bit	0	0	0	0	0	0	4,808	4,808	4,808	14.7	15.2	17.1
stack_17768_bit	0	0	0	17,768	17,768	17,768	0	0	0	14.7	15.1	15.5

Figure 5.8: Each cell gives results from the 10 runs in the format 'min median max', with all values being rounded to 3 significant figures, except for NetworkManager channel capacity. This is because all runs round to 16.0 bits (the ground truth), but none achieved exactly that figure. Note that *Channel Capacity* here is the lower-bound that is calculated by taking $\log_2$ of the number of distinct outputs for a single public input.

matches the ground truth exactly. The largest of these is the 80 bit leak from uninitialised stack memory that occurs in AppleTalk. For the other 3 programs, we still obtain estimates much higher than the provided lower-bounds and most notably a 443,960 bit estimate in the median run on Heartbleed (equating to 55,495 bytes against a 65,535 byte ground truth).

Finally, we see that for the IFSpec_ programs, this channel capacity estimate is somewhat ineffective. This is due to the processing steps between input and output resulting in less direct correspondence between input and output bits. Nonetheless, this does provide justification for computing both the lower-bound and direct mapping metrics simultaneously, as either can be more appropriate dependending on the leak.

RQ3: How does the estimate on conditional mutual information perform on a program with known input distribution?

We provide a proof-of-concept level evaluation of CMI estimation, enabling the NIFuzz flag to force uniform selection of public inputs. This allows us to make good estimates of programs with paths of vastly differing weight, such as the following which is used in our evaluation here:

```
1 typedef unsigned int uint32;
2 uint target_func(uint32 public, uint32 secret) {
3   if (public % 10 == 0) {
4     if (secret < 0xFFFF) return 0;
5     else return 1;
6   } else {
7     return 101;
8   }
9 }
```

Here, 1 in 10 public values will be part of the set of *violations*; and for these $\frac{\texttt{0xFFFF}}{\texttt{UINT32_MAX}}$ $(= \frac{1}{65537})$ values of secret will result in a return value of 1, while the rest return 0. As a result, the mutual information between secret input and public output, conditioned on public input is just 0.0000266 bits (3sf).

The chart in figure 5.9 shows 3 hours worth of fuzzing of the above program (during which time ≈ 750 million inputs were generated and tested). The steep, near vertical peaks are caused by over estimation of leakage whenever a new violation is found; early on in the campaign (towards the left of the chart) these have a larger effect.

Conditional mutual information could be applied to testing differential privacy applications, or network traffic analysis. In the case of network traffic analysis, certain publicly readable aspects of say packet size and router destination, act as public input; while the contents (and/or final destination if a VPN is in use) constitute the secret input.

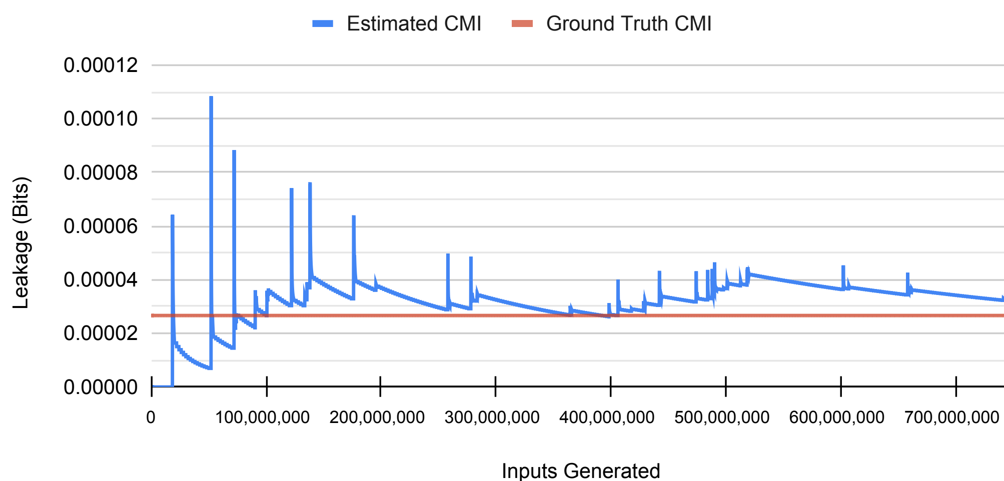


Figure 5.9: Line chart with the blue line showing the estimated CMI value as samples are collected. The red line shows the ground truth value.

5.10 Threats to Validity

All fuzzing campaigns were run with identical seeds; some were provided by the software developers or original fuzzing harness authors, and where these were not available we used a single simple generic seed that was not specialised to the SUT (i.e. was constructed with no knowledge of the expected input format). It is possible that the results could differ if the generic starting seed were replaced with different starting seeds. All internal sources of non-determinism, such as seeding of the pseudorandom number generators, was kept in place. Additionally, all of the evaluation has been done with programs that contain known information leak bugs, and we have not tried to discover new, previously unknown ones. The set of programs on which we did evaluate is limited in size and scope; unfortunately such bugs are hard to find documented, and constructing harnesses for the large multi-user programs that are capable of suffering is an extremely time consuming

process.

5.11 Conclusion

In this chapter we have introduced NIFuzz, the first fuzzer capable of quantifying information leaks through program output.

We have derived a new methodology for calculating conditional mutual information in a way that minimises the amount of information that must be stored; we need only to store details for violations. This measure models an attacker that can observe *public* input (but not influence it) and *public* output.

We have described an approach for detecting whether information leaks come from explicit *secret* inputs, or uninitialised stack or heap memory. When combined with quantification, this can be useful in identifying the source of the programming error. We provide an explore and exploit schedule for first finding, and then maximising and quantifying information leaks. Additionally, we introduce algorithms for detecting direct correspondence between *secret* input and *public* output bits, whereby flipping one results in the flipping of the other. This trades off accuracy, for reasonable estimation of huge leaks that would be infeasible to quantify through enumeration of outputs.

Finally, we evaluate NIFuzz, which implements all of the above, on a diverse set of information leaks. Here we see that it is able to find all of the known information leaks, and provide reasonable lower- and upper-bounds on their quantities – even for Heartbleed which consists of 278k LoC and leaks over

500k bits.

Chapter 6

Improving the Exploration Efficiency of Grey-Box Fuzzing – PrescientFuzz

6.1 Introduction

Fuzzing is an effective testing technique for finding bugs and vulnerabilities in software. Grey-box fuzzing leverages program coverage to more effectively explore program functionality than the early black-box fuzzing approaches. Nonetheless, even the most effective fuzzer cannot find bugs in code that its generated inputs never cover. In this chapter, we propose an additional layer of fuzzer feedback, making use of knowledge of the target program's semantics – specifically, the structure of its control flow graph (CFG) – to improve the rate of coverage discovery.

We detail the design of an input scheduler. This selects inputs for mutation based on heuristics about the number of uncovered basic blocks that border its execution path; and we then expand this idea to include the complete set of all (statically) reachable uncovered blocks within the CFG. Our approach provides the fuzzer itself with full knowledge of the CFG. The metrics used by the scheduler are recomputed repeatedly at runtime, meaning that it adapts dynamically as new coverage is achieved during the fuzzing campaign. This is somewhat similar to recent *directed* grey-box fuzzing approaches. Unlike that line of research we do not employ a set of fixed targets, but merely seek to optimise coverage as rapidly as possible. The dynamic approach also means that we are able to deal with indirect branching, as branch targets for each individual execution can be determined by examining the coverage feedback; this is vital for object-oriented software that makes significant use of dynamic dispatch.

We create an implementation of the described scheduler and build it into a fuzzer, *PrescientFuzz*, which we evaluate on the FuzzBench benchmark suite [86]; here it achieves more program coverage across the aggregated set of benchmarks than any other publicly listed fuzzer.

The feedback mechanism is implemented using an LLVM compiler pass, so is immediately available to target programs written in languages with an LLVM front-end, such as C, C++, Julia, Kotlin and Rust. The scheduler as described is generic enough that it could be implemented for any grey-box fuzzer which should see immediate benefits.

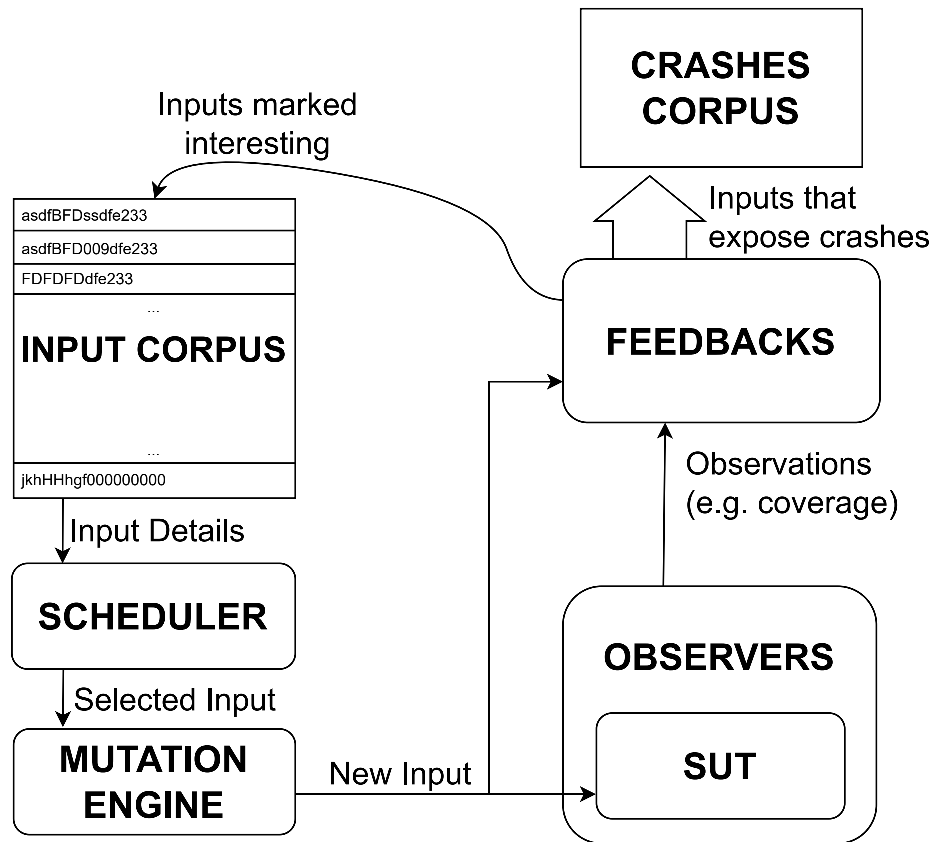


Figure 6.1: Typical architecture of a LibAFL-based fuzzer

Finally, we believe that our feedback approach has additional benefits. For example it can be used to improve the heuristics for determining when to end a fuzzing campaign. When doing concolic fuzzing, it could also help determine the best conditionals to solve in order to maximise exploration potential.

6.2 Generalisable Techniques in Fuzzing

In this section, we introduce a common architecture for modern fuzzers, and discuss a number of techniques that have proven useful enough to be

adopted into the current state-of-the-art grey-box fuzzers. Like our approach, these techniques are abstract enough to be applicable to many fuzzers, rather than being a particular implementation specific optimisation.

Figure 6.1 shows the architecture of a LibAFL-based fuzzer. Note that LibAFL offers the ability to build black-, grey- and white-box fuzzers; the diagram omits the symbolic execution runtime and solver required for white-box fuzzing. There are a number of different avenues to be explored when it comes to improving the performance of a fuzzer; the input scheduler, mutation engine, observers and feedback. For example, the difference between black-box and grey-box fuzzers is the addition of a coverage observer (and feedback). The following paragraphs list some well-known approaches that have been applied successfully in the past, with an aim to demonstrate how the different architectural components can be modified. Unlike all of these, our approach combines static knowledge of the SUT's semantics (the control flow graph of the SUT), with dynamic coverage feedback, to create a new advantage.

AFLFast [19] extended AFL with 'power schedules', in essence moving AFL from its original round-robin *scheduler* to one of several specified in the paper. Of the six proposed schedules, "Fast" is the most commonly used and has been re-implemented in both mainline AFL++ and LibAFL.

Redqueen [9] is the name of a grey-box fuzzer that aims to help pass difficult conditional checks. It does so by searching for 'input to state correspondence'; that is, relationships between parts of the input and the conditional check values. To do this, it provides an additional *observer* that records the

values of any non-constant comparands at conditional checks, and the *feedback* is passed on to the *mutation engine*, which can try inserting the values into the input. For example, when fuzzing a png decoder without a valid png file as seed, the first obstacle is the 8 byte magic number expected at the very start of the file. Redqueen is able to detect that these 8 bytes from the input are used in a comparison, and tries replacing these with the value of the other comparand. Versions of this approach are also implemented in mainline AFL++ and LibAFL.

MOpt [76] is a technique that improves the *mutation engine* by optimising the schedule for applying mutation operators, prioritising those that have been working well so far in the campaign. This can be very effective for certain types of program. The commonly used set of mutators include random bitflips, inserting constant values and splicing. For something like a programming language parser, random bitflips are likely to be less useful than splicing; and MOpt can capitalise on this. Again, this technique has been re-implemented in mainline AFL++ and LibAFL.

Directed grey-box fuzzing is an idea introduced by AFLGo [20]; here a set of code points are marked as targets, and the fuzzer aims to reach these. This is particularly useful for reproducing bugs when given access only to a stack trace, and testing patches. In order to do so, distances are computed at compile time indicating the least number of edges between a given basic block and any targets – inputs that get closer to the target are prioritised for mutation. The distance acts as a form of *feedback*, and the *scheduler* is adapted

to prioritise inputs based on this. Our approach bears similarities to directed grey-box fuzzing in that it uses distance metrics computed from the SUT's (System Under Test) control flow graph, however it differs in that it has no specific target at which it is being 'directed'. Instead, it aims to explore as much of the SUT's functionality as possible by maximising total coverage.

Sanitizers introduce additional *feedback* by extending the error detection oracle that the fuzzer has. To do this they build some sanity checks into the SUT that causes it to crash if any are violated. For example, AddressSanitizer [107] can find memory errors such as use-after-free, buffer overflows and memory leaks; and MemorySanitizer can find uninitialised memory reads. These sanitizers are implemented by compiling additional checks into the SUT.

Fishfuzz [132] is a directed grey-box fuzzer that uses sanitizer checks as goals; this allows it to aim for code areas that are more likely to trigger a crash in the SUT. It has proven to be an effective approach for discovering bugs.

6.3 Our Approach

Here we describe the approach used by *PrescientFuzz* for scheduling inputs for selection from the corpus. Note that the specific details of our implementation are discussed separately in Section 6.4.

To quote the Fuzzbench paper [86], which introduces a comprehensive bench-

mark set for evaluating fuzzers: *A fuzzer can only detect a bug if first it manages to cover the code where the bug is located.* Thus we propose a technique to improve the rate at which coverage is discovered; particularly at the early stages of fuzzing campaigns.

To illustrate the utility of our approach we now present a worked example. Suppose we have a function that we wish to fuzz as follows:

```
1  int example(int in1, int in2, int in3) {
2      int res = 0;
3      if (in1 > 15) {
4          res = 1;
5      } else if (in1 < 2) {
6          res = 2;
7      } else if (in1 < 4) {
8          res = 3;
9      } else if (in1 < 8) {
10         res = 4;
11     } else { // 8 < i <= 15
12         res = 5;
13     // Hard check to break through
14     if (in1 ^ in2 == 0xDEADBEEF) {
15         switch (in3) {
16             case 0: res = 6; break;
17             case 1: res = 7; break;
18             case 2: res = 8; break;
19             case 3: abort(); break;  // crash the program!
20             default: res = 9; break;
21         }
22     }
23
24     return res;
25 }
```

Figure 6.2: An example program in the C programming language

The control flow graph for this particular example is shown in figure 6.3:

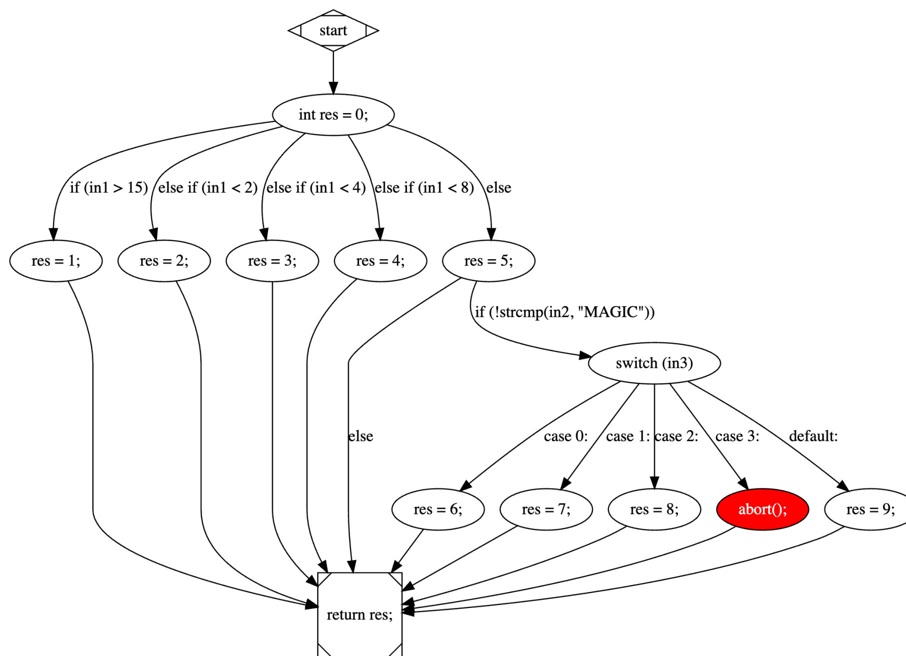


Figure 6.3: Control flow graph for figure 6.2. Note the crash at abort, colored red.

Assuming the use of a typical coverage-guided grey-box fuzzer, let's suppose we first discover the branch leading from `int res = 0;` to `res = 1;`, followed by discovery of the branch to `res = 2;`, perhaps by using an input that inserts the commonly occurring constant value 0 into variable `in1`. We then go on to discover the other 3 possible blocks reachable from `int res = 0;`. The covered blocks are marked green in the following:

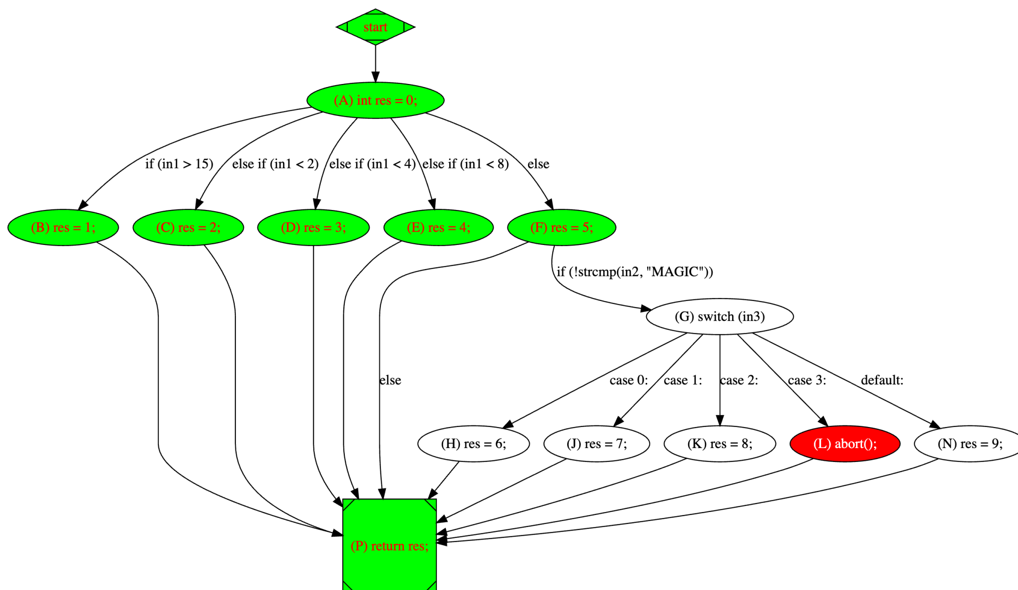


Figure 6.4: A control flow graph for the program in figure 6.2, with covered blocks filled green.

A common fuzzer strategy, employed in both LibAFL and AFL++ is to select a subset of the input corpus that achieves 100% coverage of the currently discovered blocks; this subset is internally referred to as 'favored'. It is selected by sorting the list of corpus entries in such a way that the fastest executing, shortest inputs (calculated by `execution_time * input_length`) are first; and then greedily selecting inputs from this sorted list until all current covered blocks are represented. Favored inputs are more likely to be selected for mutation; in this case, all 5 inputs would be marked favored (as all 5 are required to cover all edges discovered), which is reasonable. Additional gains can be achieved by weighting the selection probability for each input in the corpus. This is what *PrescientFuzz* does, but using a novel weighting algorithm explained in the remainder of this section.

6.3.1 Background Concepts

We assume that at a given point in the fuzzer's campaign, the minimum (dynamic, updated) state of the fuzzer that we *must* consider is the set of inputs in the corpus and the set of basic blocks that have been covered to date. There are further elements of the state that could be considered, for instance the exact priority assigned to elements of the input corpus by the scheduler, but these two are sufficient dynamic information to define the underlying idea of our approach.

This relies in turn on two ideas: The set of uncovered basic blocks in the CFG that are reachable from a given covered basic block, and the set of minimal length, loop free paths where the first basic block in the path has already been covered and all subsequent blocks in the path are uncovered. These represent the available fresh territory that can be reached from that discovered already. The CFG is the map of the territory and the executions of the target SUT are the means by which we explore this map.

In the following we provide a formal definition of the concepts used by *PrescientFuzz*. Illustrative explanations of some of these concepts are provided in subsequent subsections.

Let CFG denote a control flow graph of a given program P, defined as a tuple CFG $= (BB, Edges)$, where:

- BB is the set of basic blocks in the CFG. A basic block is defined as a straight line code section containing no branches (except for any incoming branches at the beginning and any outgoing branches at the end).
- $Edges \subseteq BB \times BB$ represents the set of transitions between blocks, with each transition indicating possible control flow from one block to another. Note that the control flow graph is a directed graph.

Given a specific input, we assume that executing the program with this input will always result in the same set of blocks being covered.

- I is the complete input space
- $i \in I$ is an input belonging to the input space
- $P(i)$ is the path in the CFG obtained by executing the program P on input i. If the input has been executed, the basic blocks in the path are termed *covered*.
- $Cov(i) \subseteq BB$ is the set of basic blocks in $P(i)$

For a fuzzing campaign input corpus $C, C \subseteq I$, define:

- $AllCov(C) = \bigcup_{i \in C} Cov(i)$

- DUN is short for Direct Uncovered Neighbour:

$$DUN(bb) = \{x \in BB \mid \exists Edge(bb, x) \ \wedge \ bb \in BB \ \wedge$$
$$Edge \in Edges(bb, x) \ \wedge \ x \notin AllCov(C)\}$$

- $ADUN$ is short for All Direct Uncovered Neighbours.

$$ADUN(Cov(i)) = \{DUN(bb) \mid bb \in Cov(i)\}$$

$ADUN(Cov(i))$ is then the set of uncovered basic blocks that are one-step reachable from the set of blocks in the CFG that correspond to the execution path for input i.

Let $R(start, end) \subseteq Edges$ be a minimal length sequence of transitions (considered as a Route) in the control flow graph to get from $start$ to end. Overload $R(start, end)$ to mean also the set of basic blocks in the route. Assume $R(start, end)$ is a non-empty ordered pair with the left basic block the starting point and the right basic block the end point, with individual transitions between them represented as ordered pairs (bb_i, bb_{i+1}). Define:

- $start$ as a specific starting block $\in BB$,
- end as a specific ending block $\in BB$.

Then, $R(start, end)$ must satisfy the following conditions:

1. The first transition in $R(start, end)$ must start from $start$, i.e., if (bb_1, bb_2)

is the first element of $R(start, end)$, then $bb_1 = start$.

2. Each subsequent transition must start at the end of the previous transition, and there are no cycles.

3. The last transition in $R(start, end)$ must end at end, i.e., if (bb_k, bb_{k+1}) is the last element of $R(start, end)$, then $bb_{k+1} = end$.

As above, we overload $R(start, end)$ to stand for the set of blocks traversed on the route, as well as the set of edges on the path, the difference being clear from context.

Now define:

- RUB is short for Reachable Uncovered Blocks. These are the uncovered basic blocks that can be reached from a given covered block, bb, in some transition sequence of uncovered blocks.

$$RUB(bb) = \Big\{x \in BB \mid \exists R(bb, x) \land \forall (s, e) \text{ in } R(bb, x),$$
$$\{s, e\} \cap (AllCov(C) \setminus \{bb\}) = \emptyset\Big\}$$

- $ARUB$ is short for All Reachable Uncovered Blocks.

  Let $X \subseteq Cov(C),$ then $ARUB(X) = \bigcup_{bb \in X} RUB(bb)$

6.3.2 Direct Uncovered Neighbours

The intuition behind our approach is to use the knowledge of the control flow graph to select inputs for mutation, prioritising based on the number of uncovered blocks that can be reached in the CFG from the execution path

for the input. We define a *direct uncovered neighbour* as being an uncovered block that is reachable from the current path by following a single edge, but is not in the set of blocks covered by any input in the corpus.

Taking for example the control flow graph shown in figure 6.4. If we have an input covering the edges $\{A, B, P\}$, we see that from A it is possible to reach $\{B, C, D, E, F\}$, and from B it is possible to reach only P. So we have a set of *possible direct uncovered neighbours* of $\{B, C, D, E, F, P\}$. If we first eliminate blocks that have been covered by this input, we are left with $\{C, D, E, F\}$. If we also exclude the other covered edges marked in green – $\{C, D, E, F\}$ – we are left with the empty set $\{\}$.

If instead we take an input covering $\{A, F, P\}$, we have a set of *possible direct uncovered neighbours* $\{B, C, D, E, F, G, P\}$. Eliminating the blocks that have been covered leaves us with one *direct uncovered neighbour*: G.

A naive approach would be to give a weighting equal to the number of *direct uncovered neighbours*, which from the current state leaves all paths apart from $\{A, F, P\}$ with a weight of 0, and $\{A, F, P\}$ itself with a weight of 1. This would mean that we would only ever select the latter input (covering $\{A, F, P\}$) for mutation. Hopefully it is intuitive why this is a good approach for achieving more coverage; and a fuzzer cannot find bugs in code that it does not cover.

6.3.3 Reachable Blocks

We extend the concept of *direct uncovered neighbours*, for an input i, to be the set of any blocks that are reachable from the set of blocks covered when executing i, without needing to traverse any blocks that have already been covered by other inputs during the fuzzing process — we call this set the *reachable blocks*.

To find the set of *reachable blocks* from a given set of blocks covered by executing an input i, we perform a breadth-first search starting with each of the blocks covered by i added to the queue, and the complete set of blocks covered by *all* inputs added to visited. We keep track of the depth at which each *reachable block* was visited. Pseudocode describing this process is as follows:

```
1  func calc_reachable_blocks(fuzzer, input):
2      queue = Queue()
3      depth = 0
4      covered = fuzzer.execute(input).covered_blocks
5      for block in covered:
6          queue.push_back( (block, depth) )
7
8      visited = HashSet()
9      for block in fuzzer.all_covered_blocks:
10         visited.insert(block)
11
12     reachable_blocks = []
13
14     while not queue.is_empty():
15         (block, depth) = queue.pop_front()
16         for successor in block.successors:
17             if not visited.contains(successor):
18                 visited.insert(successor)
19                 reachable_blocks.append( (successor, depth + 1) )
20                 queue.push_back( (successor, depth + 1) )
21
22     return reachable_blocks
```

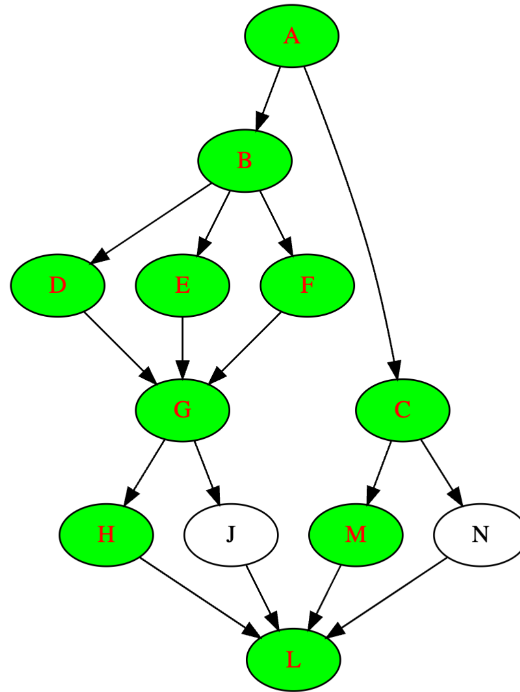


Figure 6.5: A control flow graph for a program, with covered blocks filled green.

For the example program in Figure 6.2, with input covering $\{A, F, P\}$, starting with

`fuzzer.all_covered_blocks` being `[A, B, C, D, E, F, P]` the resultant value of `reachable_blocks` is `[(G, 1), (H, 2), (J, 2), (K, 2), (L, 2), (N, 2)]`.

6.3.4 Rarity Weighting

We use the rarity weighting to attempt to equalise the amount of fuzzing effort that is put towards reaching each *reachable block*. Take for example the CFG shown in Figure 6.5. Again, we use a green fill to indicate nodes that have been covered. Let's say we have inputs in the fuzzing corpus that cover the following paths, and *reachable blocks*:

```
Path                    Reachable Blocks

[A, B, D, G, H, L]      {(J, 1)}

[A, B, E, G, H, L]      {(J, 1)}

[A, B, F, G, H, L]      {(J, 1)}

[A, C, M, L]            {(N, 1)}
```

Directly taking the number of reachable blocks as the weighting, every one of these inputs would have a score of 1. This means that 3 out of 4 times we would select an input for mutation that has J as reachable. There is no particular reason that we should spend 3 times the amount of effort on attempting to reach that block as opposed to N. Thus we propose to normalise the effort, by keeping track of the number of times each block appears in the sets of *reachable blocks* and giving a score proportional to the inverse of this as follows:

```
reachability_frequency = {} # empty dictionary
# Calc the number of times we see each reachability
for input in fuzzer.corpus:
    reachable = calc_reachable_blocks(fuzzer, input)
    for (block, depth) in reachable:
        freq = reachability_frequency[(block, depth)]
        if freq is None:
            freq = 0
        reachability_frequency[(block, depth)] = freq + 1

for input in fuzzer.corpus:
    # Here, weighting is the probability of an input
    # being selected for mutation
    weighting = 0
    reachable = calc_reachable_blocks(fuzzer, input)
    for (block, depth) in reachable:
        weighting += 1 / reachability_frequency[(block, depth)]
```

Using this new calculation, we get the following:

```
reachability_frequency = {
    (J, 1): 3,
    (N, 1): 1
}
```

```
Path                Reachable Blocks  Weighting
[A, B, D, G, H, L]  {(J, 1)}          1 / 3
[A, B, E, G, H, L]  {(J, 1)}          1 / 3
[A, B, F, G, H, L]  {(J, 1)}          1 / 3
[A, C, M, L]        {(N, 1)}          1 / 1
```

Using these weightings, we now have an equal probability of selecting an input for mutation that borders either J or N.

6.3.5 Depth Weighting

Additionally, we weight the *reachable blocks* based on their *depth*, which is the minimum number of edges that need traversing to reach the given block. Again, we use an inverse for the weighting: $1 \div$ depth. Note that the instrumentation we use does not instrument dominated or post-dominated blocks, thus in practice the depth is actually the number of conditionals that need to be passed to reach a given block. Figure 6.6 illustrates the blocks that are actually instrumented after this minimisation step; these nodes are colored blue:

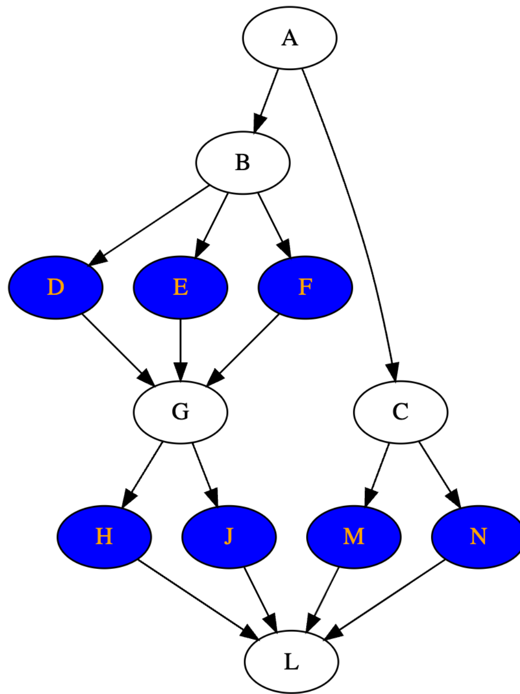


Figure 6.6: A control flow graph showing the minimal set of nodes (filled blue) that need instrumenting in order to infer the complete set of nodes covered by a program execution

6.3.6 Weighted Fuzzer Corpus Scheduling

Combining all of the elements described previously, we get the following score calculation:

```
1  func compute_score(reachability_frequency, input):
2      reachability_score = 0
3      reachable = calc_reachable_blocks(fuzzer, input)
4      for (block, depth) in reachable:
5          # Using inverse depth prioritises more immediately reachable
6          # blocks
7          score = 1 / depth
8          # if many inputs have the same reachability, weight it less;
9          # thus inputs with rarely seen reachable_blocks are more
10         # likely to be chosen
11         score *= 1 / reachability_frequency[(block, depth)]
12         reachability_score += score
13
14     # input.exec_time() is the amount of time taken to execute this
15     # input, slower inputs get a lower score
16     time_score = 1 / input.exec_time()
17
18     return reachability_score * time_score
```

```
1  func compute_all_scores(fuzzer):
2      reachability_frequency = {}
3      # Compute how many times we see each reachability
4      for input in fuzzer.corpus:
5          reachable = calc_reachable_blocks(fuzzer, input)
6          for (block, depth) in reachable:
7              freq = reachability_frequency[(block, depth)]
8              if freq is None:
9                  freq = 0
10             reachability_frequency[(block, depth)] = freq + 1
11
12     input_weightings = {}
13     for input in fuzzer.corpus:
14         score = compute_score(reachability_frequency, input)
15         input_weightings.insert(input, score)
16
17     return input_weightings
```

Note that because the initial set of covered blocks (`fuzzer.all_covered_blocks` in `calc_reachable_blocks`) changes every time that new coverage is

found (i.e. any new coverage is *added* to it), the complete set of scores must be recalculated whenever a new entry discovering coverage is added to the corpus. For some of the benchmarks that we evaluated on, there were over 20,000 basic blocks and over 1,000 inputs in the corpus; in these cases `compute_all_scores` has excessive overhead.

In order to mitigate this, we store the amount of time that `compute_all_scores` takes to run, and only allow it to be run again once $10\times$ that amount of time has elapsed (we refer to this as a *cooldown period*). If new coverage is discovered in the meantime, a flag is set to indicate that the scores need recomputing; this flag is checked whenever a new input is to be selected for mutation. At the beginning of the fuzzing campaign, it is possible that many inputs achieving new coverage are discovered during the mutation of a single seed – the scores are only used for selecting the next input for fuzzing, hence do not need recalculating until we actually go to select a new input.

During the *cooldown period*, any new inputs that are added to the corpus are assigned a score equal to the average of all other inputs in the corpus. We see that at the beginning of a fuzzing campaign there are typically many inputs discovered during the cooldown period, but afterwards the discovery rate slows enough that each new discovery results in an immediate recalculation when the next corpus entry is selected.

The most similar approach, FishFuzz, pre-computes a static distance between functions and they give the argument that fully computing basic block distances would be too costly; we have not found this to be the case with the

implementation of the *cooldown period*. Our dynamic approach has a key advantage in that indirect branches can be resolved at runtime by looking at the coverage feedback for each input; indirect branching is common in object-oriented languages, where dynamic dispatch is used. A further result of using dynamic computations, is that we could alternate between directed grey-box fuzzing targets at runtime if so desired; whereas this would typically require costly re-compilation of the SUT. In some ways, we have unintentionally built a *universal* directed grey-box fuzzer.

6.4 Implementation

In this section, we describe the implementation specific details of our approach as it is used in *PrescientFuzz*. We found that storing the complete control flow graph (CFG), in such a way as to be able to align it with the coverage feedback received from individual executions, to be a complex endeavour; we believe that this is the reason that our relatively simple idea has not been done already. We implemented a version of the technique using LibAFL [42], and use a custom LLVM compiler pass in order to store the CFG details to a file as part of the SUT's compilation process. This file is read in at runtime by the fuzzer. Note that as this is a more general technique, it is not built on any of the work of the previous chapters, which were specifically aimed at detection information leaks. That said, this technique could be adopted into NIFuzz and would likely improve its results.

In particular, this is done using a modified version of the `SanitizerCoverage`

pass that is built in to LLVM; specifically we override the `trace-pc-guard` coverage instrumentation. `trace-pc-guard` provides user-defined callback when the SUT is initialised, that gives pointers to the beginning and end of an array where each element maps to a particular basic block in the CFG. There is an additional separate callback to a user-defined function when entering each block in the SUT's CFG; this callback provides (as an argument) a pointer to one of the elements of the aforementioned array. It is expected that during the initialisation callback, the user will assign a unique value to each array element, so that during the latter callback they can establish which block has been entered.

Instead of leaving it to the user-defined initialisation callback to assign values to each edge, we preset these values during the compiler pass. As we discovered, it is likely that the reason for requiring an initialisation callback to allow the user to assign unique values to each block at runtime, is because of the difficulty in keeping track of which values have already been assigned during the compilation process. As it is common in a C/C++ build to compile each file to a `.o` object file separately, and then link these later, some state must be retained by the compiler between invocations. We overcame this by storing a counter in the CFG output file that we were producing to pass in to the fuzzer, and beginning the block enumeration at the value following on from that last assigned. Note that it is possible to register a link-time optimisation (LTO) pass with LLVM, at which point all edges can be labelled in a single pass; unfortunately this severely limits the LLVM versions that can

be simultaneously supported, so we chose not to do this.

Now we come to the file format for the CFG file itself. Firstly, we discuss the required information for each edge in the CFG. During the compiler pass, we iterate through all functions, and within these each basic block, and finally each instruction within the basic block. As shown in Figure 6.6, `trace-pc-guard` does not instrument every block in the CFG; only the minimal subset required in order to determine the exact set of covered blocks. We chose to include all blocks in our CFG file representation, whether assigned an identifier or not; for this reason, we assign each block a unique identifier that is separate to it's index in the coverage map.

For each basic block, we therefore store: a unique block identifier, the coverage map index (if one has been assigned), a list of functions (names) called within the block, a list of the unique identifiers for the *successor* blocks, and the number of indirect function calls. The list of function calls is populated whilst iterating through the instructions that make up the block; whilst we could attempt to directly list the entry block of the function as a *successor*, definitions that are in other files cannot be resolved yet. Additionally, we found that being able to determine when the fuzzer has entered a new function by simply looking at the list of called functions to be a useful debugging aid. We do not know the target of indirect function calls at compilation time, however we track these and in our reachability calculations assume that an indirect function call discovers one new basic block. This may not be the case in practice, as the indirect call may target a function that has al-

ready been explored, but in SUTs that make use of indirection this was a better approximation than ignoring them. During the LLVM pass, one can determine whether a `CallInst` is indirect using the `IsIndirectCall()` member function.

For each function, we store: the function name and a list of the unique identifiers for each basic block within the function. A pseudocode description of the pass is shown in Figure 6.7.

The CFG file is updated rather than overwritten at each stage; thus before starting the build process, the file should be cleared by the user if it has unwanted contents. Additionally, as build systems such as `make` typically compile object files in parallel, we use a lock-file (the file system equivalent to a mutex) to ensure that there is no contention over `latest_coverage_map_index`; that is, the lock is acquired in `fetch_from_CFG_file` and released at the end of `update_CFG_file`. The file contents themselves are essentially a binary serialisation of the arguments provided to the `update_CFG_file` function call.

6.5 CFG File Parsing and CFG Reconstruction

Our fuzzer takes, as an argument, a path to the CFG file. Firstly, the file is deserialised in order to reconstruct the original data structures. From here, we simplify the CFG, and cache as much information as possible to aid our reachability algorithm during the fuzzing campaign – as this needs to be ran

```
1  struct BBInfo {
2      ptr: BasicBlock *,
3      uid: int,
4      coverage_map_index: int,
5      called_funcs: String[],
6      successor_bbs: BasicBlock *[],
7      num_indirect_calls: int
8  }
9
10 latest_coverage_map_index, lastest_block_uid = fetch_from_CFG_file()
11 bbs_in_function_named = {}
12 bb_infos = []
13
14 for function in module:
15     bbs_in_func: BBInfo[] = []
16     for bb in function:
17         bb_info = BBInfo()
18
19         for succ in bb.successors:
20             bb_info.successor_bbs.append(succ)
21
22         bb_info.uid = latest_block_uid
23         latest_block_uid += 1
24
25         if bb.needs_instrumenting:
26             bb.insert_sancov_callbacks()
27             bb_info.coverage_map_index = latest_coverage_map_index
28             latest_coverage_map_index += 1
29         else:
30             bb_info.coverage_map_index = -1
31
32         for inst in bb:
33             if inst.isFuncCall():
34                 if inst.isIndirectCall():
35                     bb_info.num_indirect_calls += 1
36                 else:
37                     func_name = inst.get_called_function().name()
38                     bb_info.called_funcs.append(func_name)
39         bbs_in_func.append(bb_info)
40
41     func_name = function.name()
42     bbs_in_function_named[func_name] = bbs_in_func
43
44 update_CFG_file(
45     latest_coverage_map_index, latest_block_uid,
46     bb_infos, bbs_in_function_named
47 )
```

Figure 6.7: A pseudocode description of our compiler pass

potentially millions of times, and is computationally expensive.

In a first pass, we compute the set of directly reachable blocks from the entry of each function; we cache these so that they do not need to be recomputed every time a function call is encountered. Next, for each basic block that has been instrumented with a callback, we compute and cache the set of directly reachable blocks that have also been instrumented, this includes any blocks within function calls made on the way. This saves us from needing to traverse the uninstrumented blocks later on during the reachability search. Finally, as we assign `uid`s consecutively, we store the basic block information into an array, where the index at which it is stored corresponds to the `uid`; which saves us a level of indirection when traversing the graph.

One additional issue to overcome, is the multiple definition of functions. As our compiler pass does not run at link-time – when all definitions would have been resolved – we find that for some SUTs there are multiple definitions of some functions. We could attempt to statically walk the CFG and resolve these, however any functions reachable only by indirect calls would be tricky (if not impossible) to resolve. Instead we store lists of all definitions each function, then assume that the first in the list is the one used; if we later receive coverage feedback that indicates this was incorrect, we swap the actual encountered definition to be first in the list.

One unplanned benefit of not using the link-time pass to create the CFG is that we also receive CFG information for shared libraries, which would otherwise be excluded as they are compiled as separate modules.

6.6 General Fuzzer Implementation

The fuzzer itself is modified from LibAFL's *fuzzbench* fuzzer (note that while LibAFL is a library of fuzzing components, it also includes some example fuzzers), and as such inherits the included state-of-the-art functionality based on RedQueen [9] and MOpt [76]. Unlike LibAFL's original *fuzzbench* fuzzer, we do not use the corpus scheduler based on AFLFast [19], but our own as described in section 6.3.6.

6.7 Evaluation

In order to evaluate *PrescientFuzz*, we aimed to answer the following research questions:

RQ1: How does the coverage achieved by *PrescientFuzz* compare to other state-of-the-art fuzzers?

RQ2: How does the proposed scheduler compare to random scheduling and state-of-the-art weighted scheduling?

RQ3: How do the different weighting mechanisms described in Sections 6.3.2, 6.3.3, 6.3.4 and 6.3.5 affect the rate of coverage discovery?

To answer these questions, we evaluated *PrescientFuzz* on the standard FuzzBench coverage benchmark suite [86]. It is a collection of 22 real-world programs (and 1 artificial program – `bloaty_fuzz_target`), and the benchmarks offer significant enough diversity that results generalise well. More in depth

Benchmark	dict	# seeds	# edges
bloaty_fuzz_target	false	94	89,530
curl_curl_fuzzer_http	false	31	62,523
freetype2_ftfuzzer	false	2	19,056
harfbuzz-hb-shape-fuzzer	false	58	10,021
jsoncpp_jsoncpp_fuzzer	true	0	5,536
lcms_cms_transform_fuzzer	true	1	6,959
libjpeg-turbo_libjpeg_turbo_fuzzer	false	1	9,586
libpcap_fuzz_both	false	0	8,149
libpng_libpng_read_fuzzer	true	1	2,991
libxml2_xml	true	0	50,461
libxslt_xpath	true	112	34,860
mbedtls_fuzz_dtlsclient	false	1	10,942
openssl_x509	true	2,241	45,989
openh264_decoder_fuzzer	false	1	17,443
openthread_ot-ip6-send-fuzzer	false	0	17,932
proj4_proj_crs_to_crs_fuzzer	true	44	6,156
re2_fuzzer	true	0	6,547
sqlite3_ossfuzz	true	1,258	45,136
stb_stbi_read_fuzzer	true	166	5,026
systemd_fuzz-link-parser	false	6	53,453
vorbis_decode_fuzzer	false	1	5,022
woff2_convert_woff2ttf_fuzzer	false	62	10,923
zlib_zlib_uncompress_fuzzer	false	0	875

Table 6.1: Statistics for the 23 programs that make up the default FuzzBench coverage benchmark suite. Note that the 'dict' column indicates whether the program comes with a dictionary which helps with input generation.

details can be found in Table 6.1. Under standard setup parameters, each program is run 20 times for 23 hours by each fuzzer being evaluated; to save on running costs, instances can be pre-empted, so in many cases less than 20 runs will complete the full 23 hours. The service is generously maintained and operated free-of-charge by Google, and all fuzzers are subject to the same standards.

RQ1: How does the coverage achieved by *PrescientFuzz* compare to other state-of-the-art fuzzers?

As the unique methods implemented by *PrescientFuzz* aim to improve input scheduling, and not the mutation engine itself, we would expect any gains in program coverage to come early in the fuzzing campaign. Table 6.2 shows the relative performance for a set of fuzzers; the numerical value is the median code coverage percentage of the fuzzer across all benchmarks (relative to the best performing fuzzer on each – thus a fuzzer that performed best on all benchmarks would score 100). Note that the displayed results are made up the top-5 set of 'default' fuzzers; these are fuzzers that are actively maintained and represent the state-of-the-art. One of the big advantages of using FuzzBench is that each fuzzer has been setup by their maintainers, hence are more likely to be optimal setups. To provide some context, aflplusplus (AFL++) can build with many clang or GCC versions, but a very specific setup including the clang linker and archiver are required to get maximum performance from it – in an independent evaluation it is likely that someone would miss this and produce non-representative results. Notably, FishFuzz is missing from this evaluation as there is no provided setup for FuzzBench; though given that it targets bugs that are detectable by sanitizers above all else, it is not clear whether it would compare favourably on coverage metrics.

Fuzzer	**2 hours**	**23 hours**
PrescientFuzz	98.49	99.05
libafl	96.93	97.79
aflplusplus	95.69	95.45
honggfuzz	92.26	93.53
libfuzzer	88.77	91.81
afl	82.49	84.07

Table 6.2: Table showing the relative median coverage for all benchmarks (mean percentage) at 2 and 23 hours. Note that 'libafl' is the default libafl-based fuzzer submitted to FuzzBench; it uses an AFLFast-based corpus scheduler.

We have chosen to list results at the 2 hour mark, as this is where *Prescient-Fuzz* has the greatest advantage over 'libafl' (AFLFast-based scheduling); and 23 hours as this was the end time of the campaigns. As can be seen, PrescientFuzz compares favourably with the other fuzzers, and notably outperforms its parent libafl. As expected, it does better early on, with a 1.6% advantage over libafl at 2 hours, dropping to 1.3% advantage by 23 hours. As can be seen by the gap between libafl and aflplusplus, the gap between fuzzers can be relatively small.

RQ2: How does the proposed scheduler compare to random scheduling and state-of-the-art weighted scheduling?

Figure 6.8 shows the relative coverage scores over time for the various libafl-based fuzzers with more time granularity – note that these aggregated statistics take a long time to compute, hence the sparsity of data points.

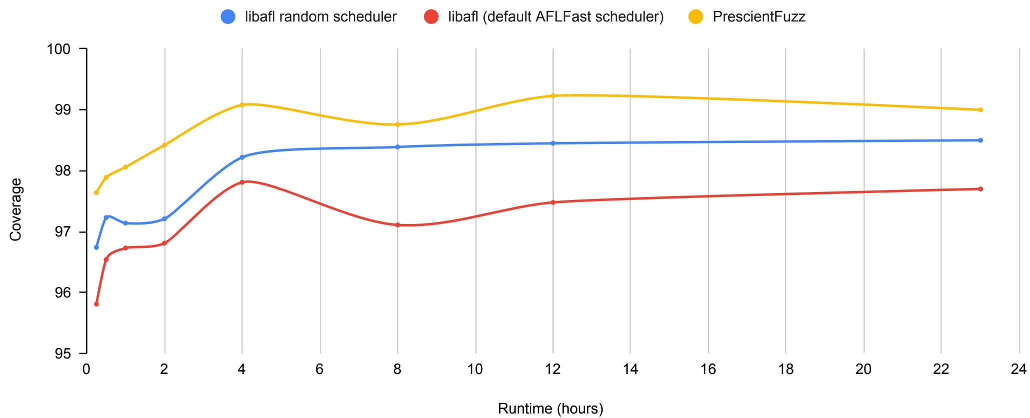


Figure 6.8: Chart showing the relative coverage for the libafl-based fuzzer setups.

As can be seen, the random scheduler outperforms libafl's default AFLFast-based scheduling approach in the long run. This was not something that we anticipated, but given that AFLFast is considered state-of-the-art for scheduling, it may mean that we have built the first scheduler to outperform random. More detailed results broken down by benchmark can be found in Table 6.3.

Benchmark	Scheduler			Vargha Delaney	
	PrescientFuzz	AFLFast-based	Random	PF–Fast	PF–Rand
bloaty_fuzz_target	6,284	6,305.5	6,346.5	0.5	0.31
curl_curl_fuzzer_http	10,748.5	10,874	10,854	0.04	0.12
freetype2_ftfuzzer	11,621.5	11,225	11,621.5	0.7	0.5
harfbuzz_hb-shape-fuzzer	11,081	11,074	11,113	0.55	0.36
jsoncpp_jsoncpp_fuzzer	517	517	517	0.62	0.53
lcms_cms_transform_fuzzer	2,139.5	2,076.5	2,104	0.77	0.62
libjpeg-turbo_libjpeg_t...	3,079	3,078	3,078	0.77	0.74
libpcap_fuzz_both	2,840	2,770	2,675	0.75	0.8
libpng_libpng_read_fuzzer	1,999.5	1,999	1,998	0.71	0.61
libxml2_xml	15,688	15,625	15,620	0.85	0.77
libxslt_xpath	11,125	10,974	11,018	0.97	0.91
mbedtls_fuzz_dtlsclient	3,552	3,105	3,416.5	0.82	0.62
openh264_decoder_fuzzer	9,440	9,446	9,475	0.47	0.33
openssl_x509	5,830	5,823	5,822	0.78	0.74
openthread_ot-ip6-send-fuzzer	3,559	3,548	3,573	0.64	0.53
proj4_proj_crs_to_crs_fuzzer	7,427	7,202.5	7,399.5	0.91	0.63
re2_fuzzer	2,856	2,856.5	2,858	0.51	0.44
sqlite3_ossfuzz	20,798	20,714	20,883.5	0.62	0.32
stb_stbi_read_fuzzer	2,197	2,140.5	2,187	0.81	0.76
systemd_fuzz-link-parser	239	237	237	0.56	0.58
vorbis_decode_fuzzer	1,252.5	1,250	1,253	0.70	0.49
woff2_convert_woff2tt_fuzzer	1,186.5	1,184.5	1,178.5	0.64	0.76
zlib_zlib_uncompress_fuzzer	449.5	450.5	450	0.42	0.45

Table 6.3: Edge coverage broken down by program. Note that 'median' here refers to the number of edges found in the median run (when sorted by coverage), and 'AFLFast-based' refers to the default `libafl` scheduler. The 'PF–Fast' column gives the Vargha-Delaney A12 measure between PrescientFuzz and AFLFast-based schedulers, and the 'PF–Rand' column gives the same measure between PrescientFuzz and the Random scheduler.

RQ3: How do the different weighting mechanisms described in Sections 6.3.2, 6.3.3, 6.3.4 and 6.3.5 affect the rate of coverage discovery?

To answer this question, we compare the aggregated coverage for all of the different scheduling setups over time; this is shown in Figure 6.9. Here the fuzzer name for the various `prescientfuzz` setups indicates which features were enabled. The `prescientfuzz_direct_neighbours` setup has scheduling based only on the number of direct uncovered neighbours, as described in section 6.3.2. The `prescientfuzz_reachable*` setups have scheduling based on the total number of reachable uncovered blocks as described in section 6.3.3. The `prescientfuzz_reachable_rarity*` setups additionally weight the schedule based on rarity as described in section 6.3.4, and finally `prescientfuzz_reachable_rarity_depth` also uses the depth metric from section 6.3.5 (this is referred to previously as just `PrescientFuzz` as it contains all of the features).

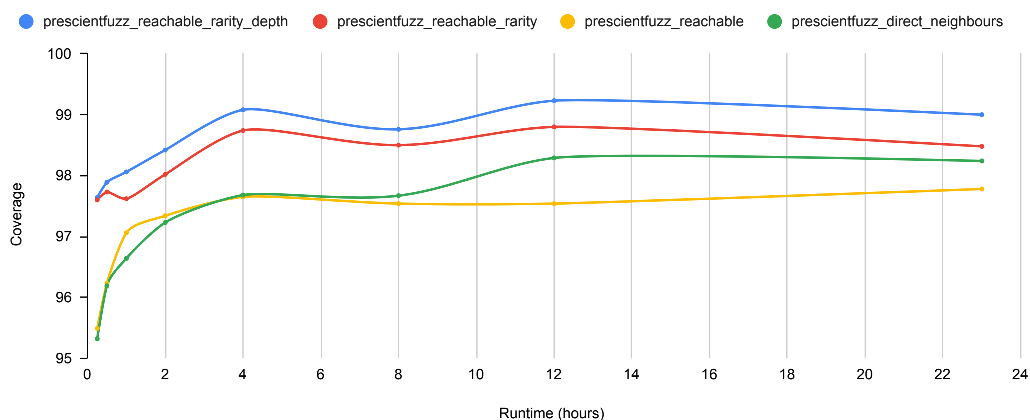


Figure 6.9: Chart showing the relative coverage achieved with different scheduling calculation steps included.

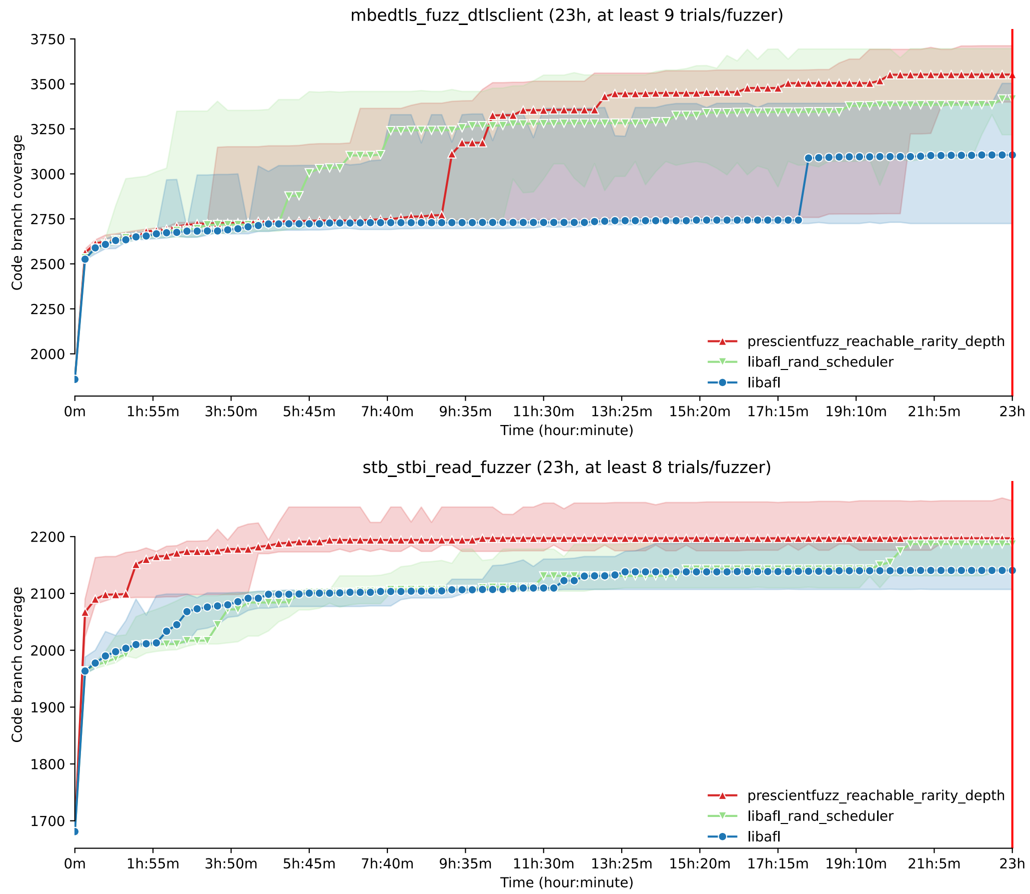


Figure 6.10: Coverage growth over time for the 2 programs where significant improvement can be seen due to *PrescientFuzz*'s scheduling. The solid line indicates the median coverage, while the shaded area encompasses the 25th and 75th percentiles.

6.8 Discussion

There were two programs where a significant improvement could be seen due to the scheduling approach taken by *PrescientFuzz*; graphs of coverage over time for these are shown in figure 6.10.

In the case of `mbed_fuzz_dtlsclient`, we see that while 'libafl_rand_scheduler' makes the initial breakthrough from the plateau around 2,600 edges earlier than 'prescientfuzz' (with median at approximately 5.5 hours into the campaign as opposed to 9 hours); the latter continues making more progress to-

wards the end of the time limit. The AFLFast-based schedule used by 'libafl' underperforms here compared to random – having just tested against this default setup at first, we believed 'prescientfuzz' to have made a more significant impact.

In `stb_stbi_read_fuzzer`, it can be seen that 'prescientfuzz' achieves median coverage of approximately 2,070 edges within 15 minutes (points on the plot are at 15 minute intervals); it takes the alternative schedulers over 2 hours to achieve the same coverage. In this case, the random scheduler catches up towards the end.

The most interesting observation from our point of view is that the random scheduler outperformed the default AFLFast-based setup. The implementation used in LibAFL's *fuzzbench* fuzzer only mutates a subset of the testcases; these are selected to cover all blocks seen in the campaign while minimising the execution time and input lengths (formerly known as 'favored' testcases in AFL). It is possible that the reduced diversity due to this pruning causes the decrease in rate of coverage discovery.

Figure 6.11 shows four programs where a slight early gain in coverage is seen for *PrescientFuzz*. In the case of `harfbuzz_hb-shape-fuzzer`, the random scheduler ultimately goes on to slightly outperform *PrescientFuzz*. We believe that the flattening of the asymptote, and the number of edges that it occurs at, is due to the power of the mutation engine. In the case of RedQueen, feedback about the program's internal state – the value of variables used in conditionals – essentially acts a powerful hint, 'turbocharging'

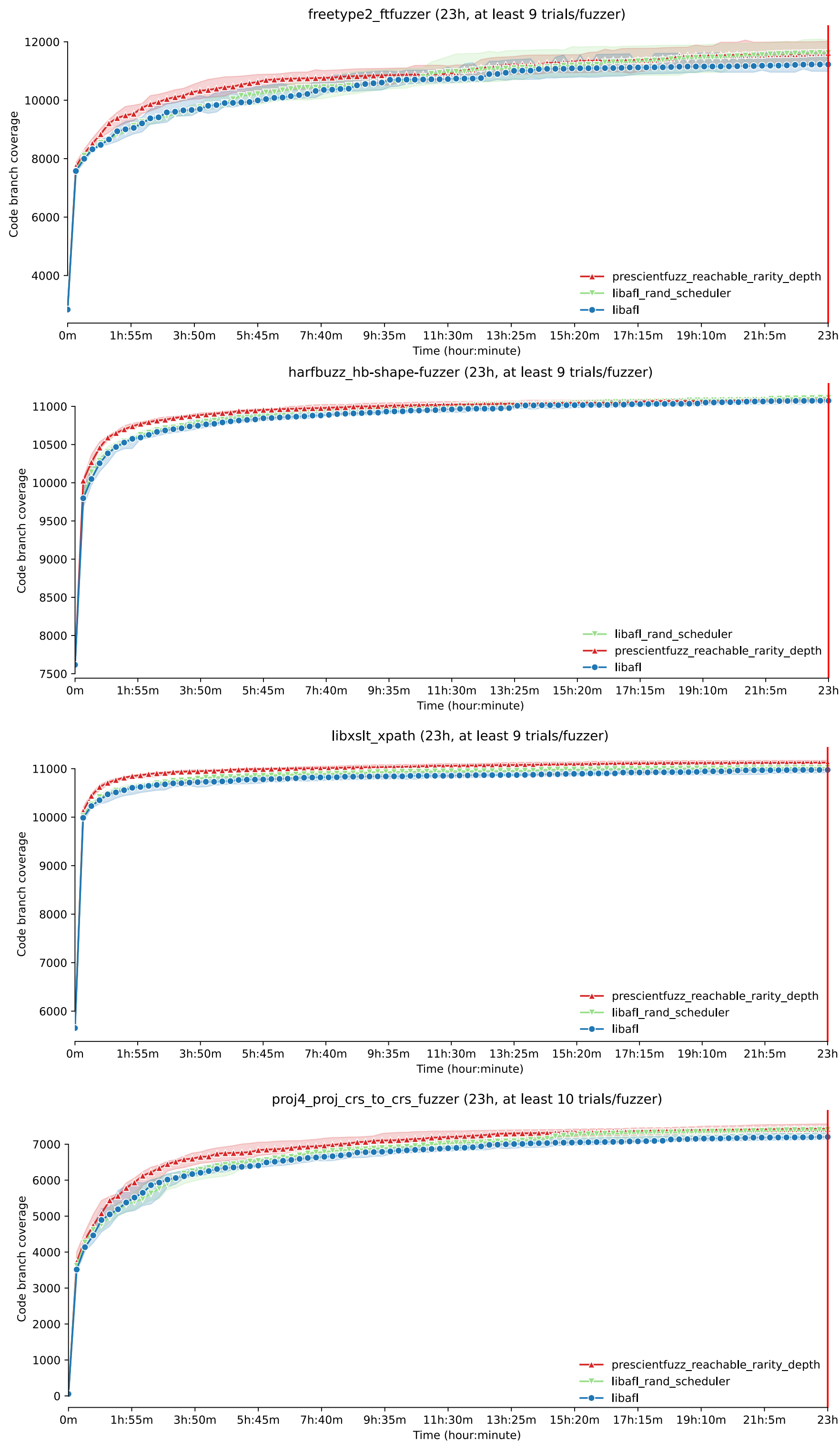


Figure 6.11: Coverage growth over time for four benchmarks where smaller early improvements can be seen due to *PrescientFuzz*'s scheduling; but other scheduling approaches catch up.

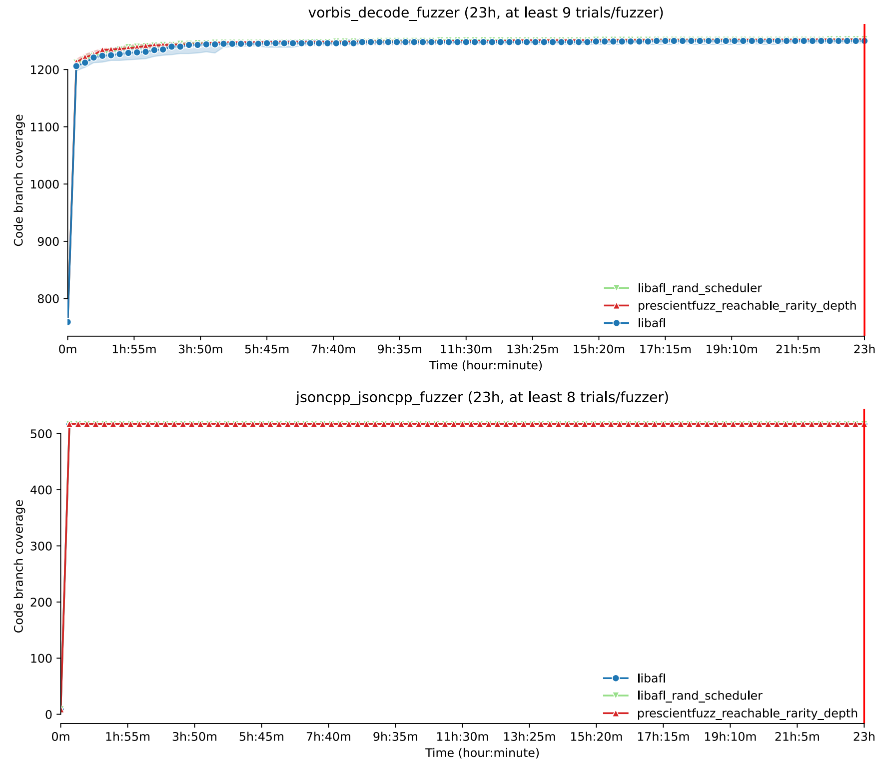


Figure 6.12: Two benchmarks where there was very little difference observed between schedulers at the 15-minute tick granularity of FuzzBench. Notice that jsoncpp (bottom) is completely flat from 15 minutes onwards.

the mutation engine. As *PrescientFuzz* does not modify the mutation engine nor provide any additional feedback that could guide it, we did not expect an improvement to the number of edges reached when the asymptote flattens – only an improvement to the time at which the flattening is reached.

Finally, Figure 6.12 shows coverage growth for two SUTs where schedule has little impact. Notice that in the `jsoncpp` benchmark, the asymptote has almost completely flattened out by 15 minutes; the time that the first measurement was taken. We observed that there were no benchmarks where *PrescientFuzz* lagged behind in median coverage at the 23 hour mark by more than 1.0%.

6.9 Impact

PrescientFuzz happens to be the best at discovering coverage on the FuzzBench set of benchmarks, but that is mainly due to 'standing on the shoulders of giants'; LibAFL's 'fuzzbench' (upon which *PrescientFuzz* is built) was previously the best performing fuzzer. We see the real value of this work as the *control flow graph* feedback and scheduling mechanism; which can be applied to any grey-box fuzzer. Given that the compiler pass is already written for LLVM, programs written in C, C++, Rust, and any other languages for which an LLVM front-end exists can benefit from our implementation immediately.

The reachability metric as described in section 6.3.3—in particular the set of *All Reachable Uncovered Blocks* (ARUB in Section 6.3.1)—could be used to provide feedback to help determine when to end a fuzzing campaign; improving the estimates provided by some papers [74, 17]. For example, seeing that coverage discovery has stalled but there still exists a large area of undiscovered coverage (indicated by a large number of reachable blocks) may mean that it is worth continuing to fuzz. If instead coverage discovery has stalled, but the only remaining uncovered blocks are at depth 1 (i.e. are direct neighbours to covered blocks) then it may be wiser to end the fuzzing campaign.

Additionally, the reachability metric could be useful in determining which branching conditions are best candidates to be solved for when using con-

colic fuzzing. Concolic fuzzing uses a concrete input to build up a set of constraints that are required to reach a certain program point; essentially a formula describing the necessary features of the input required. We can foresee that sorting the set of basic blocks for each input by the number of reachable blocks – Then creating and solving the formula for the block with most reachability – may vastly improve the efficiency, and get the most value out of each expensive solve.

6.10 Threats to Validity

Outsourcing our evaluation to the hosted FuzzBench service means that we had no control over the execution conditions; despite this, FuzzBench is commonly trusted for the evaluation of fuzzers in academic work [42, 18, 80].

6.11 Conclusion

In this chapter we have introduced a new dynamic feedback mechanism for grey-box (and white-box) fuzzers, that makes use of the SUT's CFG semantics. We also document the design of an input corpus scheduler, that combines this knowledge from the CFG with coverage feedback from the corpus itself, in order to select inputs for mutation that have a higher probability of discovering new coverage. We built a grey-box fuzzer, *PrescientFuzz*, and evaluated it on FuzzBench, where it achieves the most coverage across the aggregated set of benchmarks of any publicly available fuzzer. Finally, we

discussed how the feedback mechanism could be used to improve decisions about when a fuzzing campaign should be halted, and how it could be used by concolic fuzzers in order to improve the leverage of the computationally expensive solving process by intelligently selecting which conditionals to solve.

Chapter 7

Future Work

For all the work that is presented in this thesis on fuzzing and information flow control, there is still much to be explored. In this chapter I explore three ideas that I believe to be the good candidates for future pursuit.

7.1 Applying the Principles of Data-Flow Guided Fuzzing to Hypercoverage

Hypercoverage is a new idea, presented in a 2024 paper [97]. They consider hypercoverage in terms the set of basic-blocks in which a *low* output variable is assigned. They give the following program as an example:

```
1  if (key == 0) {
2      log = key + 5;
3  } else {
4      if (log == 0) {
5          log = 5;
6      } else {
7          if (key > 6) {
8              log = key;
9          } else {
10             log = 0;
11             key = 1;
12         }
13     }
14 }
```

Here, `log` is considered output; hence the basic-blocks at line ranges `{2}`, `{5}`, `{8}`, `{10-11}` are all part of the set that can make up hypercoverage. Note that due to the control flow structure, only one of these basic-blocks will be executed in every program execution.

The authors propose two search-based software testing techniques for finding violations to hyperproperties – one based on EvoSuite [43], and one fuzzing.

I believe that the concept of hypercoverage could improve NIFuzz's violation discovery abilities. In particular, I propose a metric based on the number of conditional checks that are dependent on the *secret* part of the program input. This dependency can be detected using DataFlowSanitizer, a taint analysis tool. Once one has determined which bytes from a given input a conditional depends upon, we can use knowledge from the input's structure to decide whether any of these bytes belong to the *secret* part of the input. Take, for example, the following function (program):

```
1  int myLeakyFunc(int secret, int public) {
2      int res = 0;
3      if (public > 10) {
4          bool secretIsBig = secret > 100;
5          if (secretIsBig && public % 5 == 0) {
6              res = 1;
7          } else if (secret < 10) {
8              res = 2;
9          } else {
10             res = 3;
11         }
12     }
13     return res;
14 }
```

Due to the use of dynamic taint analysis, my proposed metric is local to each particular input. For example, given that the input `{secret: 0, public 0}` fails the first conditional check, it encounters only one conditional check; and `public > 10` is independent of the value of *secret*. For the input `{secret: 20, public: 0}` three conditionals are encountered: `public > 10`, `secretIsBig && public % 5 == 0` and `secret < 10`. Of which, the latter two are dependent on the value of *secret* (note that `secretIsBig` is transitively dependent on `secret > 100`). Given that there are three paths that could be potentially be taken, and two of these depend on the value of *secret*, we can infer that mutating this input is more likely to yield a failing hypertest than the former test.

There are flaws with the incomplete information provided by this approach: we still do not know whether a given conditional is feasible; whether there exist some unsatisfied parts of the condition which depend on the *public* part of the input; or whether the *public* output is modified by the path behind the condition. Despite these shortcomings, this metric could still be useful in

guiding the search for failing hypertests. Additionally, concolic execution can be used strategically to resolve the former two issues.

7.2 New Targets for NIFuzz

Whilst both LeakFuzzer and its successor NIFuzz have both been evaluated on a diverse set of programs, and proven themselves capable of detecting known information flow control violations; their ability to detect new, previously unknown, ones can only be demonstrated by doing such. I believe that to encourage adoption and to highlight the prevalence of such issues, this utility needs to be demonstrated. As I have discovered, one of the major difficulties with creating appropriate fuzzing harnesses for software is the lack of specifications of flow security policies, and difficulties in deriving them where they are not available. Additionally, the multi-user systems that make for good targets are typically very large and complex, with so much functionality that significant expertise is needed in order to create good harnesses; for example the `PostgreSQL` bug that LeakFuzzer was able to detect uses obscure functionality from the 900+kLoC codebase.

My proposal to alleviate these issues would be to work directly with maintainers of the software that is being tested. Maintainers will have good oversight of the codebase, and are likely to be able to point out areas of functionality that have the potential to leak confidential information. They will also be well placed to provide reasonable security policies, if not full specifica-

tions. I see three good candidates to target: Tor [38], PostgreSQL [1] and the Linux Kernel [2].

7.3 Improving Fuzzer Corpus Diversity

Improving fuzzer efficiency is beneficial not just to the likes of LeakFuzzer and NIFuzz, but fuzzing in general. We saw during the evaluation of PrescientFuzz (Section 6.7) that the random scheduler outperformed the heuristic-based AFLFast type scheduler for the LibAFL fuzzer. Additionally, AFL++ saw significant improvements in disabling the deterministic mutation set used by default in the original AFL; instead applying mutations at random. Finally, in some recent literature [82, 103] it has been observed that increasing the 'diversity' of the input corpus can be beneficial to grey-box fuzzers.

To me, all of these suggest a correlation between the diversity of the inputs generated by a grey-box fuzzer and its effectiveness; of course there is a balance, and too much diversity would devolve into black-box fuzzing. Given that the mutation engine starts with inputs taken from the corpus, I believe that there may be improvements to be found in exploring approaches to maximising diversity within that corpus whilst maintaining the target program coverage.

[1] https://www.postgresql.org/

[2] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

Chapter 8

Conclusions

In this thesis on *fuzzing and information flow*, we have introduced a number of key techniques for both the discovery and estimation of the quantity of insecure flows in real-world scale software.

In Chapter 4, we introduced the concept of hyperfuzzing – a fuzzer-based approach to hypertesting, and evaluated LeakFuzzer, a hyperfuzzer that detects violations of the input-output non-interference property. Unlike formal methods based approaches to detecting violations of non-interference, LeakFuzzer can scale to programs of arbitrary size, and in the evaluation was able to find hypertests demonstrating violations of security policies for programs with 94k, 279k and 905k lines of code. One of the contributing factors to the scalability in the face of huge input spaces, was the caching of a map between *public* inputs and *public* outputs; this means that rather than having to construct and evaluate all hypertests (pairs of *public-private* input pairs),

of which there are $\approx n^2$ for an input space of size n, each *public-private* input pair needs evaluating only once. This moves the computational complexity from $O(n^2)$ to $O(n)$ in exchange for memory usage; which itself is partially resolved by storing hashes rather than full inputs or outputs (most of the time).

In Chapter 5, we extended the concepts introduced by the work on LeakFuzzer to include quantified information flow (QIF) for any violations of security policies. We derived a formula for calculating conditional mutual information – for our usecase this is the mutual information between *secret* input and *public* output, conditioned on *public* input; modelling the viewpoint of an attacker who is able to observe *public* input but not control it. We discussed why counting the number of unique outputs possible for any *public* input is the only way to truly calculate channel capacity. However, we also introduced an algorithm for estimating huge channel capacity quantities that looked for correspondence between input and output bits – that is, flipping a bit in the input causes a bit in the output to flip also. This algorithm allowed our NIFuzz tool to quickly estimate the channel capacity for artificial programs with known ground truth up to 17,768 bits (2,221 bytes). It also estimated the leakage of Heartbleed at over 55k bytes – while the 'optimal' malicious input is capable of triggering 65k bytes of leakage, it is unlikely for a fuzzer to produce such an input.

In Chapter 6, we explored approaches for improving the efficiency of greybox fuzzing; as these were not specific to hyperfuzzing, we evaluated them

in a traditional fuzzing context. The implementation, PrescientFuzz, performed well in its' evaluation and is a generalisable technique that can be paired with existing grey-box fuzzers, and may have even more impact when paired with white-box fuzzing techniques.

Bibliography

[1] AFLPlusPlus - Shengtuo Hu (h1994st). *Grammar-Mutator*. URL: https://github.com/AFLplusplus/Grammar-Mutator.

[2] Cyan 4973. *xxHash*. https://github.com/Cyan4973/xxHash. 2014.

[3] AFLplusplus. *LibAFL, the fuzzer library*. https://github.com/AFLplusplus/LibAFL.

[4] Dakshi Agrawal et al. "The EM side—channel (s)". In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2002, pp. 29–45.

[5] Mohammad Abdullah Al Faruque et al. "Acoustic side-channel attacks on additive manufacturing systems". In: *2016 ACM/IEEE 7th international conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2016, pp. 1–10.

[6] Timos Antonopoulos et al. "Decomposition instead of self-composition for proving the absence of timing channels". In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 362–375.

[7] apple. *Swift - libFuzzer Integration*. `https://github.com/apple/swift/blob/main/docs/libFuzzerIntegration.md`.

[8] Steven Arzt et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps". In: *Acm Sigplan Notices* 49.6 (2014), pp. 259–269.

[9] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *NDSS*. Vol. 19. 2019, pp. 1–15.

[10] Gergö Barany and Julien Signoles. "Hybrid information flow analysis for real-world C code". In: *Tests and Proofs: 11th International Conference, TAP 2017, Held as Part of STAF 2017, Marburg, Germany, July 19–20, 2017, Proceedings 11*. Springer. 2017, pp. 23–40.

[11] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. "Secure information flow by self-composition". In: *Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252.

[12] D Elliott Bell and Leonard J LaPadula. *Secure computer systems: Mathematical foundations*. Tech. rep. MITRE CORP BEDFORD MA, 1973.

[13] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. Califor-nia, USA. 2005, pp. 10–5555.

[14] Fabrizio Biondi et al. "HyLeak: hybrid analysis tool for information leakage". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2017, pp. 156–163.

[15] Fabrizio Biondi et al. “Scalable approximation of quantitative information flow in programs”. In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer. 2018, pp. 71–93.

[16] Daniel Blackwell, Ingolf Becker, and David Clark. “Hyperfuzzing: black-box security hypertesting with a grey-box fuzzer”. In: *arXiv preprint arXiv:2308.09081* (2023).

[17] Marcel Böhme. “STADS: Software testing as species discovery”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27.2 (2018), pp. 1–52.

[18] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. “Boosting fuzzer efficiency: An information theoretic perspective”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 678–689.

[19] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. “Coverage-based greybox fuzzing as markov chain”. In: *IEEE Transactions on Software Engineering* 45.5 (2017), pp. 489–506.

[20] Marcel Böhme et al. “Directed greybox fuzzing”. In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 2329–2344.

[21] Robert S Boyer, Bernard Elspas, and Karl N Levitt. “SELECT—a formal system for testing and debugging programs by symbolic execution”. In: *ACM SigPlan Notices* 10.6 (1975), pp. 234–245.

[22] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. “Jvm fuzzing for jit-induced side-channel detection”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 1011–1023.

[23] Niklas Broberg, Bart van Delft, and David Sands. “Paragon for practical programming with information-flow control”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2013, pp. 217–232.

[24] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. “HLIO: Mixing static and dynamic typing for information-flow control in Haskell”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 2015, pp. 289–301.

[25] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.” In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[26] Jia Chen, Yu Feng, and Isil Dillig. “Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 875–890.

[27] Eric Cheng. “Binary Analysis and Symbolic Execution with angr”. PhD thesis. PhD thesis, The MITRE Corporation, 2016.

[28] Haehyun Cho et al. "Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. 2020.

[29] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. "A Tool for Estimating Information Leakage". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 690–695. ISBN: 978-3-642-39799-8.

[30] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. "LeakWatch: Estimating Information Leakage from Java Programs". In: *Computer Security - ESORICS 2014*. Ed. by Mirosław Kutyłowski and Jaideep Vaidya. Cham: Springer International Publishing, 2014, pp. 219–236. ISBN: 978-3-319-11212-1.

[31] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 2000, pp. 268–279.

[32] David Clark, Sebastian Hunt, and Pasquale Malacaria. "A static analysis for quantifying information flow in a simple imperative language". In: *Journal of Computer Security* 15.3 (2007), pp. 321–371.

[33] M. R. Clarkson and F. B. Schneider. "Hyperproperties". In: *21st IEEE Computer Security Foundations Symposium*. 2008.

[34] Byron Cook et al. "Model checking boot code from AWS data centers". In: *Computer Aided Verification: 30th International Conference, CAV*

2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30. Springer. 2018, pp. 467–486.

[35] Cover and Thomas. *Elements of Information Theory*. Second. Wiley, 2006.

[36] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[37] Dorothy E Denning. "A lattice model of secure information flow". In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

[38] Roger Dingledine, Nick Mathewson, Paul F Syverson, et al. "Tor: The second-generation onion router." In: *USENIX security symposium*. Vol. 4. 2004, pp. 303–320.

[39] Ulfar Erlingsson and Fred B Schneider. "SASI enforcement of security policies: A retrospective". In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. Vol. 2. IEEE. 2000, pp. 287–295.

[40] Andrea Fioraldi et al. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[41] Andrea Fioraldi et al. "Dissecting american fuzzy lop: a fuzzbench evaluation". In: *ACM transactions on software engineering and methodology* 32.2 (2023), pp. 1–26.

[42] Andrea Fioraldi et al. "Libafl: A framework to build modular and reusable fuzzers". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 1051–1065.

[43] Gordon Fraser and Andrea Arcuri. "Evosuite: automatic test suite generation for object-oriented software". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 416–419.

[44] Christian Fritz et al. "Highly precise taint analysis for android applications". In: (2013).

[45] Patrice Godefroid, Michael Y Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing". In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[46] Joseph A Goguen and José Meseguer. "Security policies and security models". In: *1982 IEEE Symposium on Security and Privacy*. IEEE. 1982, pp. 11–11.

[47] Google. *Atheris: A Coverage-Guided, Native Python Fuzzer*. 2022. URL: https://github.com/google/atheris.

[48] google. *afl-based-fuzzers-overview.md*. https://github.com/google/fuzzing/blob/master/docs/afl-based-fuzzers-overview.md.

[49] google. *fuzzer-test-suite*. https://github.com/google/fuzzer-test-suite.

[50] google. *honggfuzz*. https://honggfuzz.dev/.

[51] google. *OSS-Fuzz - Integrating a Go project*. https://google.github.io/oss-fuzz/getting-started/new-project-guide/go-lang/.

[52] google. *OSS-Fuzz: Continuous Fuzzing for Open Source Software*. 2022. URL: https://github.com/google/oss-fuzz.

[53] google. *sanitizers*. https://github.com/google/sanitizers.

[54] google. *syzkaller - kernel fuzzer*. https://github.com/google/syzkaller.

[55] Andreas Gornik et al. "A hardware-based countermeasure to reduce side-channel leakage: Design, implementation, and evaluation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.8 (2015), pp. 1308–1319.

[56] GraphicsFuzz. *secure and reliable graphics drivers*. https://www.graphicsfuzz.com/.

[57] Tobias Hamann et al. "A uniform information-flow security benchmark suite for source code and bytecode". In: *Nordic Conference on Secure IT Systems*. Springer. 2018, pp. 437–453.

[58] Shaobo He, Michael Emmi, and Gabriela Ciocarlie. "ct-fuzz: Fuzzing for Timing Leaks". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020, pp. 466–471.

[59] Jonathan Heusser and Pasquale Malacaria. "Quantifying information leaks in software". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. 2010, pp. 261–269.

[60] Sam Hocevar. *zzuf - multi-purpose fuzzer*. `http://caca.zoy.org/wiki/zzuf`. 2007.

[61] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2012. URL: `https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf`.

[62] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. "POSTER: AFL-based Fuzzing for Java with Kelinci". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2511–2513.

[63] Yoongu Kim et al. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 361–372.

[64] Johannes Kinder. "Hypertesting: The case for automated testing of hyperproperties". In: *3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot)*. Citeseer. 2015.

[65] James C King. "A new approach to program testing". In: *ACM Sigplan Notices* 10.6 (1975), pp. 228–233.

[66] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[67] Vladimir Klebanov, Norbert Manthey, and Christian Muise. "SAT-based analysis and quantification of information flow in programs". In: *Quantitative Evaluation of Systems: 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings 10*. Springer. 2013, pp. 177–192.

[68] George Klees et al. "Evaluating fuzz testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2123–2138.

[69] Paul Kocher et al. "Spectre attacks: Exploiting speculative execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1–19.

[70] Daniel Kroening and Michael Tautschnig. "CBMC–C bounded model checker". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2014, pp. 389–391.

[71] Caroline Lemieux and Koushik Sen. "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 475–485.

[72] Zhengchuan Liang et al. "K-LEAK: Towards Automating the Generation of Multi-Step Infoleak Exploits against the Linux Kernel". In: *31st Annual Network and Distributed System Security Symposium, NDSS*. 2024.

[73] Moritz Lipp et al. “Meltdown: Reading kernel memory from user space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 973–990.

[74] Danushka Liyanage et al. “Extrapolating Coverage Rate in Greybox Fuzzing”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–12.

[75] LLVM. *libFuzzer – a library for coverage-guided fuzz testing.* `https://llvm.org/docs/LibFuzzer.html`.

[76] Chenyang Lyu et al. “{MOPT}: Optimized mutation scheduling for fuzzers”. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1949–1966.

[77] David R MacIver, Zac Hatfield-Dodds, et al. “Hypothesis: A new approach to property-based testing”. In: *Journal of Open Source Software* 4.43 (2019), p. 1891.

[78] Pasquale Malacaria, Michael Tautchning, and Dino DiStefano. “Information leakage analysis of complex C code and its application to OpenSSL”. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2016, pp. 909–925.

[79] Valentin JM Manès et al. “The art, science, and engineering of fuzzing: A survey”. In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2312–2331.

[80] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. "Fuzzing with data dependency information". In: *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2022, pp. 286–302.

[81] Björn Mathis et al. "Detecting information flow by mutating input data". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 263–273. DOI: 10.1109/ASE.2017.8115639.

[82] Hector D Menendez and David Clark. "Hashing fuzzing: introducing input diversity to improve crash detection". In: *IEEE Transactions on Software Engineering* (2021).

[83] Ziyuan Meng and Geoffrey Smith. "Calculating bounds on information leakage using two-bit patterns". In: *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*. PLAS '11. San Jose, California: Association for Computing Machinery, 2011. ISBN: 9781450308304. DOI: 10.1145/2166956.2166957.

[84] Ibrahim Mesecan et al. "HyperGI: Automated Detection and Repair of Information Flow Leakage". In: *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021.

[85] Ibrahim Mesecan et al. "Keeping Secrets: Multi-objective Genetic Improvement for Detecting and Reducing Information Leakage". In: *37th IEEE/ACM International Conference on Automated Software Engineering, ASE*. 2022.

[86] Jonathan Metzman et al. "FuzzBench: an open fuzzer benchmarking platform and service". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1393–1403.

[87] Barton P Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[88] Amir Moradi. "Side-channel leakage through static power". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2014, pp. 562–579.

[89] Max Moroz. *google/AFL*. `https://github.com/google/AFL/blob/master/docs/status_screen.txt`. 2019.

[90] A. C. Myers et al. *Jif: Java information flow. Software release*. `https://www.cs.cornell.edu/jif`.

[91] Andrew C Myers. "JFlow: Practical mostly-static information flow control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1999, pp. 228–241.

[92] nccgroup. *Fuzzowski*. `https://github.com/nccgroup/fuzzowski`.

[93] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[94] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. "DifFuzz: differential fuzzing for side-channel analysis". In: *2019 IEEE/ACM*

41st International Conference on Software Engineering (ICSE). IEEE. 2019, pp. 176–187.

[95] Yannic Noller and Saeid Tizpaz-Niari. "QFuzz: quantitative fuzzing for side channels". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021, pp. 257–269.

[96] Corina S Păsăreanu and Neha Rungta. "Symbolic PathFinder: symbolic execution of Java bytecode". In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010, pp. 179–180.

[97] Michele Pasqua, Mariano Ceccato, and Paolo Tonella. "Hypertesting of Programs: Theoretical Foundation and Automated Test Generation". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–12.

[98] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "T-Fuzz: fuzzing by program transformation". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 697–710.

[99] Quoc-Sang Phan and Pasquale Malacaria. "Abstract model counting: a novel approach for quantification of information leaks". In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '14. Kyoto, Japan: Association for Computing Machinery, 2014, pp. 283–292. ISBN: 9781450328005. DOI: 10.1145/2590296.2590328.

[100] Quoc-Sang Phan et al. "Symbolic quantitative information flow". In: *ACM SIGSOFT Software Engineering Notes* 37.6 (2012), pp. 1–5.

[101] rust-fuzz. *The libfuzzer-sys Crate*. `https://github.com/rust-fuzz/libfuzzer`.

[102] Andrei Sabelfeld and Andrew C Myers. "Language-based information-flow security". In: *IEEE Journal on selected areas in communications* 21.1 (2003), pp. 5–19.

[103] Nico Schiller et al. "Novelty Not Found: Adaptive Fuzzer Restarts to Improve Input Space Coverage (Registered Report)". In: *Proceedings of the 2nd International Fuzzing Workshop*. 2023, pp. 12–20.

[104] Fred B Schneider, Greg Morrisett, and Robert Harper. "A language-based approach to security". In: *Informatics*. Springer. 2001, pp. 86–101.

[105] Sergej Schumilo et al. "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2597–2614.

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

[107] Konstantin Serebryany et al. "{AddressSanitizer}: A Fast Address Sanity Checker". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 309–318.

[108] Kostya Serebryany. "{OSS-Fuzz}-Google's continuous fuzzing service for open source software". In: (2017).

[109] Kostya Serebryany. "ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety". In: (2019). https://www.usenix.org/system/files/login/articles/login_summer19_03_serebryany.pdf.

[110] Kostya Serebryany et al. "Gwp-asan: Sampling-based detection of memory-safety bugs in production". In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 2024, pp. 168–177.

[111] Kostya Serebryany et al. "Memory tagging and how it improves C/C++ memory safety". In: *arXiv preprint arXiv:1802.09517* (2018).

[112] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. "Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1275–1290. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/shen-zekun.

[113] Geoffrey Smith and Dennis Volpano. "Secure information flow in a multi-threaded imperative language". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1998, pp. 355–364.

[114] François-Xavier Standaert. “Introduction to side-channel attacks”. In: *Secure integrated circuits and systems*. Springer, 2010, pp. 27–42.

[115] François-Xavier Standaert, Tal G Malkin, and Moti Yung. “A unified framework for the analysis of side-channel key recovery attacks”. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2009, pp. 443–461.

[116] Evgeniy Stepanov and Konstantin Serebryany. “MemorySanitizer: fast detector of uninitialized memory use in C++”. In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, pp. 46–55.

[117] Nick Stephens et al. “Driller: Augmenting fuzzing through selective symbolic execution.” In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[118] Lei Sun et al. “AFLTurbo: Speed up Path Discovery for Greybox Fuzzing”. In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2020, pp. 81–91.

[119] tegansb and Daniel Schwartz-Narbonne. *github.com - diffblue/cbmc - Parsing Errors when compiling C++ #5489*. `https://github.com/diffblue/cbmc/issues/5489`. 2020.

[120] Kris Tiri et al. “A side-channel leakage free coprocessor IC in 0.18 μm CMOS for Embedded AES-based Cryptographic and Biometric Processing”. In: *Proceedings of the 42nd annual Design Automation Conference*. 2005, pp. 222–227.

[121] Saeid Tizpaz-Niari et al. “Quantitative estimation of side-channel leaks with neural networks”. In: *International Journal on Software Tools for Technology Transfer* 23.4 (2021), pp. 641–654.

[122] M Vervier et al. “Browser Security Whitepaper, Last access 05.12. 2019.[Online]. Available: h ps”. In: *browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf* ().

[123] *Virtual private network (VPN) usage in the United States from 2019 to 2022, by location [Graph]*. 2022. URL: https://www-statista-com.libproxy.ucl.ac.uk/statistics/1291880/vpn-usage-in-the-us/.

[124] Dennis Volpano and Cynthia Irvine. “Secure flow typing”. In: *Computers & Security* 16.2 (1997), pp. 137–144.

[125] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. “A sound type system for secure flow analysis”. In: *Journal of computer security* 4.2-3 (1996), pp. 167–187.

[126] Dennis Volpano and Geoffrey Smith. “A type-based approach to program security”. In: *Colloquium on Trees in Algebra and Programming*. Springer. 1997, pp. 607–621.

[127] Yingchen Wang et al. “Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86”. In: *Proceedings of the USENIX Security Symposium (USENIX)*. 2022.

[128] J Todd Wittbold and Dale M Johnson. "Information Flow in Nondeterministic Systems." In: *IEEE Symposium on Security and Privacy*. Vol. 161. 1990.

[129] Zhemin Yang and Min Yang. "Leakminer: Detect information leakage on android with static taint analysis". In: *2012 Third World Congress on Software Engineering*. IEEE. 2012, pp. 101–104.

[130] Michał Zalewski. *american fuzzy lop (2.52b)*. `http://lcamtuf.coredump.cx/afl/`.

[131] Steve Zdancewic. "Challenges for information-flow security". In: *Proceedings of the 1st International Workshop on Programming Language Interference and Dependence (PLID'04)*. 2004.

[132] Han Zheng et al. "{FISHFUZZ}: Catch Deeper Bugs by Throwing Larger Nets". In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 1343–1360.