


# AntiFuzz: Impeding Fuzzing Audits of Binary Executables

**Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz,**
***Ruhr-Universität Bochum***

https://www.usenix.org/conference/usenixsecurity19/presentation/guler

**This paper is included in the Proceedings of the 28th USENIX Security Symposium.**

**August 14–16, 2019 • Santa Clara, CA, USA**

978-1-939133-06-9

**Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.**

# AntiFuzz: Impeding Fuzzing Audits of Binary Executables

Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz
*Ruhr-Universität Bochum*

## Abstract

A general defense strategy in computer security is to increase the cost of successful attacks in both computational resources as well as human time. In the area of binary security, this is commonly done by using obfuscation methods to hinder reverse engineering and the search for software vulnerabilities. However, recent trends in automated bug finding changed the modus operandi. Nowadays it is very common for bugs to be found by various fuzzing tools. Due to ever-increasing amounts of automation and research on better fuzzing strategies, large-scale, dragnet-style fuzzing of many hundreds of targets becomes viable. As we show, current obfuscation techniques are aimed at increasing the cost of human understanding and do little to slow down fuzzing.

In this paper, we introduce several techniques to protect a binary executable against an analysis with automated bug finding approaches that are based on fuzzing, symbolic/concolic execution, and taint-assisted fuzzing (commonly known as *hybrid fuzzing*). More specifically, we perform a systematic analysis of the fundamental assumptions of bug finding tools and develop general countermeasures for each assumption. Note that these techniques are not designed to target specific implementations of fuzzing tools, but address general assumptions that bug finding tools necessarily depend on. Our evaluation demonstrates that these techniques effectively impede fuzzing audits, while introducing a negligible performance overhead. Just as obfuscation techniques increase the amount of human labor needed to find a vulnerability, our techniques render automated fuzzing-based approaches futile.

## 1 Introduction

In recent years, fuzzing has proven a highly successful technique to uncover bugs in software in an automated way. Inspired by the large number of bugs found by fuzzers such as AFL [56], research recently focused heavily on improving the state-of-the-art in fuzzing techniques [10, 11, 22, 44, 54]. Previously, it was paramount to manually remove checksums and similar roadblocks from the fuzzing targets. Additionally, fuzzers typically required large, exhaustive seed corpora or a precise description of the input format in form of a grammar. In a push towards a greater degree of automation, research recently focused on avoiding these common roadblocks [14, 39, 44, 45, 48, 54]. This push toward automation greatly simplifies the usage of these tools. One can argue that, for the attacker, using a fuzzer is as easy as it is for the defender. In fact, recently the *Fuzzing Ex Machina* (FExM) [49] project managed to reduce the overhead of running fuzzers to a degree where they managed to fuzz the top 500 packages from the Arch Linux User Repository with no manual effort in seed selection or similar issues. This two day effort yielded crashes in 29 of the most popular packages of Arch Linux. It stands to reason that this kind of indiscriminate, dragnet-style searching for software bugs will become more prevalent in the future.

While the developers of a software system should typically thoroughly fuzz test every type of software, in practice they may want to maintain an asymmetric cost advantage. More specifically, it should be easier for the maintainers of a software project to fuzz their own software than for attackers. This can be achieved by adding mechanisms to the software such that the final binary executable is protected against fuzzing: the maintainers can then build an internal version that can be tested thoroughly, while an attacker can only access the protected binary which prohibits automated tests. In the past, similar asymmetric advantages in analysis and bug finding were introduced by obfuscation techniques. As we demonstrate, even very high levels of obfuscation will typically result only in a meager slowdown of current fuzzing techniques. This is due to the fact that obfuscation typically aims at protecting against program understanding and formal reasoning. On the other hand, fuzzers typically do not perform a significant amount of reasoning over the behaviour of the program. On the downside, these heavy obfuscation mechanisms will often incur a significant runtime overhead [19].

How software can be protected against fuzzing in an efficient and effective way is an open problem.

In this paper, we tackle this challenge and present several general methods to impede large scale, automated fuzzing audit of binary executables. We present several techniques that can be added during the compilation phase of a given software project such that the resulting binary executable withstands fuzzing and significantly hampers automated analysis. Our methods are based on a systematic analysis of 19 current bug finding tools with respect to their underlying assumptions. Note that we use the terms "fuzzer" and "bug finding tool" interchangeably to describe all kinds of tools that are analyzing programs to produce crashing inputs as opposed to static analysis tools and linters. We find that all of them rely on at least one of the following four basic assumptions: (i) coverage yields relevant feedback, (ii) crashes can be detected, (iii) many executions per second are achievable, and (iv) constraints are solvable with symbolic execution. Based on these insights, we develop fuzzing countermeasures and implement a lightweight protection scheme in the form of a configurable, auto-generated single C header file that developers can add to their application to impede fuzzers. For the evaluated programs, we had to change on average 29 lines of code, which took less than ten minutes. With these changes, attackers now need to spend a significant amount of time to manually remove these anti-fuzzing mechanisms from a protected binary executable (typically magnified by common obfuscation techniques on top), greatly increasing the cost of finding bugs as an attacker. Defenders, on the other hand, can still trivially fuzz the unmodified version of their software with no additional cost. Thus, only unwanted and unknown attackers are at a disadvantage.

We implemented a prototype of the proposed methods in a tool called AntiFuzz. We demonstrate in several experiments the effectiveness of our approach by showing that state-of-the-art fuzzers cannot find bugs in binary executables protected with AntiFuzz anymore. Moreover, we find that our approach introduces no observable, statistically significant performance overhead in the SPEC benchmark suite.

**Contributions** In summary, in this paper we make the following contributions:

- We present a survey of techniques employed by current fuzzers and systematically analyze the basic assumptions they make. We find that different fuzzing approaches rely on at least one of the fundamental assumptions which we identify.
- We demonstrate how small changes to a program nullify the main advantages of fuzzing by systematically violating the fundamental prerequisites. As a result, it becomes significantly harder (if not impossible with current approaches) to find bugs in a protected program without manual removal of our anti-fuzzing methods.
- We implemented our anti-fuzzing techniques in a tool called AntiFuzz that adds fuzzing countermeasures during the compilation phase. Our evaluation with several different programs shows that with a negligible performance overhead, AntiFuzz hardens a target binary executable such that none of the tested fuzzers are able to find any bugs.

To foster research on this topic, we release our implementation and the data sets used as part of the evaluation at https://github.com/RUB-SysSec/antifuzz.

## 2 Technical Background

Fuzzing (formerly known as random testing) has been around since at least 1981 [20]. In the beginning, fuzzers would simply try to execute programs with random inputs. While executing, the fuzzer observes the behavior of the program under test: if the program crashes, the fuzzer managed to find a bug and the input is stored for further evaluation. Even though this technique is surprisingly simple—particularly when compared to static program analysis techniques—with a sufficient number of executions per second it has been helpful at finding bugs in complex, real-world software in the past.

In recent years, the computer security community paid much more attention to improving the performance and scalability of fuzzing. For example, the OSS-Fuzz project has been fuzzing many highly-relevant pieces of software 24/7 since 2016 and exposed thousands of bugs [1]. FEXM automatized large parts of the setup and the authors were able to fuzz the top 500 packages from the Arch Linux User Repository [49]. To improve the usability of fuzzers in such scenarios, the biggest focus of the research community is to automatically overcome hard-to-fuzz code constructs that previous methods could not successfully solve with the goal of reaching deeper parts of the code. Particularly, common program analysis techniques were applied to the problem of fuzzing. For example, symbolic execution and its somewhat more scalable derivative concolic execution was used to overcome hard branches and trigger bugs that are only trigger-able by rare conditions [25–27, 31, 42, 48, 50, 54]. Other fuzzers use taint tracing to reduce the search space to mutations that actually influence interesting parts of the program [14, 23, 31, 42, 45]. A complementary line of work focused on improving the fuzzing process itself without falling back to (often costly) program analysis techniques. Many techniques propose improvements to the way AFL instruments the target [2, 22, 29, 47], or how inputs are scheduled and mutated [10, 11, 13, 46, 52]. Some methods go as far as removing hard parts from the target [44, 50]. Lastly, the effectiveness of machine learning models for efficient input generation was evaluated [9, 28, 32].

Generally speaking, existing methods for fuzzing can be categorized into the following three different categories based

on the techniques employed, which we explain in more detail in the following.

## 2.1 Blind Fuzzers

The oldest class of fuzzers are so-called *blind fuzzers*. Such fuzzers have to overcome the problem that random inputs will not exercise any interesting code within a given software. Two approaches were commonly applied: *mutational fuzzing* and *generational fuzzing*.

Mutational fuzzers require a good corpus of inputs to mutate. Generally, mutational fuzzers do not know which code regions depend on the input file and which inputs are necessary to reach more code regions. Instead, these fuzzers introduce some random mutations to the file and can only detect if the program has crashed or not. Mutational fuzzers need seed files that cover large parts of interesting code as they are unable to uncover new code effectively. In the past, these fuzzers were quite successful at uncovering bugs [33]. However, they typically need to perform a large number of executions per second to work properly. An example of mutational-only fuzzers are ZZUF [5] and RADAMSA [33].

The second approach is generational fuzzing: fuzzers which employ this technique need a formal specification to define the input format. Based on this specification, the fuzzer is able to produce many semi-valid inputs. This has the advantage that the fuzzer does not need to learn how to generate well-formed input files. However, manual human effort is necessary to create these definitions (e.g., a grammar that describes the input format). This task becomes hard for complex or unknown formats and the specification could still end up lacking certain features. The additional need for a formal specification makes this approach much less useful for large-scale bug hunting with little human interaction. An example of a generational fuzzer is PEACH [3].

In summary, the only thing a blind fuzzer is able to observe is whether its input led to a crash of the program or not. Therefore, these techniques have no indicator of their progress in exploring the programs state space and thus (especially in the case of mutational fuzzers), they are mostly limited to simple bugs even with non-empty and well-formed seed files.

## 2.2 Coverage-guided Fuzzers

To improve the performance of the mutational fuzzers, Zalewski introduced an efficient way to measure coverage-feedback of an application [56]. This led to a significant amount of research on coverage-guided fuzzers. These fuzzers typically use a feedback mechanism to receive information on how an input has affected the program under test. The key idea here is that this mechanism gives means by which to judge an input: Which (new) code regions were visited and how often? In contrast, a blind fuzzer introduces random mutations to the input without knowing how those mutations affect the program. It effectively relies on pure chance for finding crashing inputs, while a coverage-guided fuzzer could mutate the same input file iteratively to increase the code coverage and thus get closer to new regions where a crash could happen. Examples of coverage-based fuzzers are AFL [56], HONGGFUZZ [4], ANGORA [14], T-FUZZ [44], KAFL [47], REDQUEEN [8] and VUZZER [45]. These fuzzers use multiple ways to obtain coverage feedback:

**Static Instrumentation:** One of the fastest methods for obtaining code coverage is static compile time coverage (widely used by tools such as AFL, ANGORA, LIBFUZZER, and HONGGFUZZ). In this case, the compiler adds special code at the start of each basic block that stores the coverage information. From a defender's point of view, this kind of instrumentation is not relevant, as we assume that the attackers do not have access to the source code.

**Dynamic Binary Instrumentation (DBI):** If only a binary executable is available, fuzzer typically use dynamic binary instrumentation (DBI) to obtain coverage information. This is done by adding the relevant code at runtime. Examples of this approach are VUZZER and STEELIX [39], which both use PIN-based [40] instrumentation, and AFL which has multiple forks using QEMU, PIN, DYNAMORIO, or DYNINST for DBI. Fuzzers like DRILLER [48] and T-FUZZ use AFL under the hood and typically rely on the QEMU-based instrumentation.

**Hardware Supported Tracing:** Modern CPUs support various forms of hardware tracing. For Intel processors, two technologies can be used: Last Branch Record and Intel-PT. HONGGFUZZ is able to utilize both techniques, while fuzzers like KAFL only support Intel-PT.

### 2.2.1 Using Coverage Information:

Different fuzzers tend to use the coverage feedback obtained in different ways. To illustrate these differences, we select two well-known coverage-guided fuzzers; namely AFL and VUZZER. We then describe how these fuzzers are using coverage information internally. It is worth noting that by choosing AFL, we are basically covering the way various other fuzzers such as T-FUZZ, ANGORA, KAFL, STEELIX, DRILLER, LIBFUZZER, WINAFL, AFLFAST [11], and COLLAFL [22] are using coverage information. All of these fuzzers (except ANGORA) use the same underlying technique for leveraging coverage information. In contrast to AFL, no other fuzzer followed the path of VUZZER in coverage information usage. However, due to the unique usage of coverage information in VUZZER, we describe it as well.

**AFL** A key factor behind the success of AFL is an efficient, approximate representation of the code coverage. To reduce the memory footprint, AFL maps each basic block transition (edge) to one index in a fixed size array referred to as the "bitmap". Upon encountering a basic block transi-

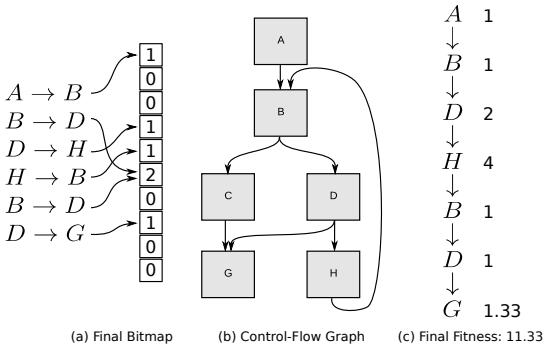



Figure 1: Using coverage information in AFL-like fuzzers versus Vuzzer in the same path of a given Control-Flow Graph (b).

tion, it increments the corresponding value in the bitmap as illustrated in Figure 1(a). The bitmap is typically limited to 64KiB, so it easily fits inside an L2 cache [55]. Although limiting the size of the bitmap allows very efficient updates, it also reduces its precision, since in some cases multiple edges share the same index in the bitmap. It is possible to increase the size of the bitmap, but at the cost of a significant decline in performance [22].

As mentioned earlier, ANGORA uses a very similar scheme with a slight difference: before updating the bitmap entry, ANGORA XORs the edge index with a hash of the call stack. This way, ANGORA can distinguish the same coverage in different contexts, while AFL can not. For example, in Listing 1, AFL cannot distinguish the coverage produced by lines 2 and 3 when called from line 10 from the coverage produced by the same lines (lines 2 and 3) in the second call. Therefore, AFL can use feedback to learn that the input should start with "fo", however, it cannot use the same information to learn that the input should continue with "ba". In contrast, ANGORA can identify the context (here "fo" and "ba") of the code and thus distinguish between these two calls. It is worth to mention that this drastically increases the number of entries in the bitmap, and therefore ANGORA might need a bigger bitmap.

Listing 1: A sample code which illustrates the differences between AFL and ANGORA on distinguishing coverage information

```
1  bool cmp(char *a, char *b){
2      if (a[0]==b[0]){
3          if (a[1]==b[1]){
4              return true;
5          }
6      }
7      return false;
8  }
9  ....
10 if (cmp(input,"fo")){
11     if (cmp(input+2, "ba")){
12     ....
13     }
14 }
```

**Vuzzer** AFL does not discriminate among edges. Therefore, an input that covers one previously unseen edge is just as interesting as an input which covers hundreds of unseen edges. This is the fundamental difference between VUZZER and AFL. Unlike AFL, VUZZER extracts the exact basic block coverage (instead of the bitmap) and enriches the feedback mechanism with additional data. For example, VUZZER uses a static disassembly to weight basic blocks according to how "deep" they are within a function (e.g., how many conditions have to be satisfied to reach this block). Higher scores are assigned to harder-to-reach blocks. To further improve the feedback mechanism, VUZZER excludes basic blocks that belong to error paths by measuring the coverage produced by random inputs. In the example shown in Figure 1(c), each basic block has a weight. As can be seen, basic block H has a much higher weight than basic block G because H is much less likely to be reached by a random walk across the control-flow graph (with back-edges removed). Finally, all the weights of all basic blocks in the path are added up to calculate a fitness value. VUZZER then uses an evolutionary algorithm to produce new mutations: inputs with a high fitness value produce more offspring. These newly created offspring are then used as the next generation.

## 2.3 Hybrid Fuzzers

While using coverage-based fuzzing already leads to interesting results on its own, there are code regions in a program which are hard to reach. This typically happens if only a very small percentage of the inputs satisfy some conditions. For example, a specific four-byte magic value that is checked by the program under the test makes it nearly impossible for coverage-based fuzzers to pass the check and therefore reach deeper code regions. To address this problem, various research suggest using a combination of program analysis techniques to assist the fuzzing process [44, 48, 54]. By using symbolic execution or taint analysis, a fuzzer is able to reason what inputs are necessary to cover new edges and basic blocks. Instead of only relying on random mutations and selection by information gathered through feedback mechanisms, these tools try to calculate and extract the correct input necessary for new code coverage. Examples of fuzzers which are using symbolic or concolic execution to assist the coverage-based fuzzer are DRILLER [48], QSYM [54], and T-FUZZ [44].

The archetypal hybrid fuzzer is DRILLER, which uses concolic execution to search for inputs that produce new coverage. It tries to provide a comprehensive analysis of the program's behaviour. In contrast, QSYM [54] identified this behavior as a weakness since the fuzzer can validate that the input proposed by the symbolic or concolic execution generates new coverage very cheaply. Therefore, an unsound symbolic or concolic execution engine can produce a large number of false positive proposals, without reducing the overall performance of the fuzzer. Building upon this insight, QSYM discards all but the last constraint in the concrete execution trace as well as the symbolic values produced by basic blocks that were executed frequently. Finally, it is worth mentioning that in the case of T-FUZZ, symbolic execution is not used for

the fuzzing process itself. Instead, T-FUZZ patches hard constraints. Once T-FUZZ finds a crashing input for the patched program, it uses symbolic execution to calculate an input that actually crashes the unpatched target program.

## 3 Analysis of Fuzzing Assumptions

Based on the categories described in the previous section, we now analyze and identify fundamental assumptions that fuzzers use to find bugs. The first insight is that while many aspects of fuzzing have changed since 1981, two basic assumptions which were originally made still apply to most modern fuzzers: these two basic original assumptions are *crash detection* and *high execution throughput*. However, to achieve better performance in modern fuzzers, additional assumptions were made in the past years, as we discuss next.

To evade not only current but also future bug finding methods, we analyze under which *core assumptions* all (or at least most) of the current tools operate. By systematically breaking assumptions shared by most fuzzers, we can develop a more universal defense against automated audits. Using this systematic approach, we avoid targeting specific implementations and therefore will hamper all future fuzzing methods built upon the same general assumptions. We divide the current fuzzing assumptions into the following four groups:

**(A) Coverage Yields Relevant Feedback** Coverage-guided fuzzers typically assume that novel code coverage also strongly correlates with novel behavior. Therefore, every time a modern coverage-guided fuzzer generates an input which traverses through a new code region, it assumes that the program behaves differently from previous inputs. Based on the coverage, the fuzzer decides how much time to allocate for generating further mutations of this input. For example, most current fuzzers such as AFL, VUZZER, DRILLER, QSYM, KAFL, ANGORA, T-FUZZ, and LIBFUZZER use this assumption for coverage-guided fuzzing.

**(B) Crashes Can Be Detected** Triggering security-relevant bugs will typically lead to a program crash. Thus, most bug finding tools need the ability to tell a crashing input apart from a non-crashing input in an efficient and scalable way. As a result, they require some techniques to detect if an application has crashed. Nearly all random testing tools share this assumption since 1981 [20]. In addition to the coverage-guided fuzzers, this assumption is also shared by blind fuzzers such as PEACH, RADAMSA, and ZZUF.

**(C) Many Executions per Second** To efficiently generate input files with great coverage, the number of executions per second needs to be as high as possible. In our experience, depending on the application and fuzzer, a range from few hundreds up to a few thousands of executions per second are typical. Slow executions will drastically degrade the performance. All fuzzers mentioned in the previous assumptions also fall into this class. Only pure symbolic execution tools such as KLEE do not fall into this category.

**(D) Constraints Are Solvable with Symbolic Execution** Hybrid fuzzers or tools based on symbolic execution such as DRILLER, KLEE, QSYM, and T-FUZZ need to be able to represent the program's behavior symbolically and solve the resulting formulas. Therefore, any symbolic or concolic execution-based tools only operate well when the semantics of the program under test are simple enough. This means that the internal representation of the state of the symbolic/concolic execution engine has to be small enough to store and the resulting constraints set has to be solvable by current solvers to avoid problems related to state explosion.

**Summary** We compiled a list of 19 different bug finding tools and systematically check which assumptions they rely on. An overview of the analyzed tools and their corresponding assumptions is shown in Table 1. It is worth mentioning that various tools in this table are based on AFL and thus share the same assumptions.

## 4 Impeding Fuzzing Audits

Based on the analysis results of the previous section, we now introduce techniques to break the identified assumptions of bug finding tools in a systematic and generic way. Moreover, we sketch how these techniques can be implemented; actual implementation details are provided in the next section.

**Attacker Model** Throughout this paper, we use the following attacker model. First, we assume that an attacker can only access the final protected binary executable and not the original source code of the software. She wants to find bugs in an automated way in the protected binary executable, while requiring only a minimum human intervention. Commonly there is the notion that source-based fuzzers significantly outperform binary-only fuzzers. Therefore, it is believed that defenders already have a significant cost advantage over attackers. However, recent advances in fuzzing have shown that this advantage is in decline. For example, recent binary-only fuzzing techniques paired with hardware acceleration technologies such as Intel PT have drastically reduced the performance gap between binary and source fuzzing. For example, Cisco Talos states that the overhead is only 5% to 15% [36] and similar numbers are reported for published Intel PT-based fuzzers such as KAFL [47]. Additionally, smart fuzzing techniques outperform source-based fuzzing even in binary-only targets [8, 54].

Although many relevant software projects are open source, a large part of all commercial software used in practice is not available in source code format (e.g., Windows, iOS and the vast majority of the embedded space). Nonetheless, some large software projects such as certain PDF viewer and hypervisors are not only well-tested by their developers, but also by whitehat attackers. This additional attention is an

Table 1: Bug finding tools and the assumptions they rely on.

| | (A) Coverage Feedback | (B) Detectable Crashes | (C) Application Speed | (D) Solvable Constraints |
|---|---|---|---|---|
| AFL | ✓ | ✓ | ✓ | ✗ |
| KAFL | ✓ | ✓ | ✓ | ✗ |
| AFLFAST | ✓ | ✓ | ✓ | ✗ |
| COLLAFL | ✓ | ✓ | ✓ | ✗ |
| AFLGO | ✓ | ✓ | ✓ | ✗ |
| WINAFL | ✓ | ✓ | ✓ | ✗ |
| STEELIX | ✓ | ✓ | ✓ | ✗ |
| REDQUEEN | ✓ | ✓ | ✓ | ✗ |
| HONGGFUZZ | ✓ | ✓ | ✓ | ✗ |
| VUZZER | ✓ | ✓ | ✓ | ✗ |
| DRILLER | ✓ | ✓ | ✓ | ✓ |
| KLEE | ✗ | ✗ | ✗ | ✓ |
| ZZUF | ✗ | ✓ | ✓ | ✗ |
| PEACH | ✗ | ✓ | ✓ | ✗ |
| QSYM | ✓ | ✓ | ✓ | ✓ |
| T-FUZZ | ✓ | ✓ | ✓ | ✓ |
| ANGORA | ✓ | ✓ | ✓ | ✗ |
| RADAMSA | ✗ | ✓ | ✓ | ✗ |
| LIBFUZZER | ✓ | ✓ | ✓ | ✗ |

important factor in their security model. Similarly, projects that have a history of helpful interactions with independent researchers should consider not to use ANTIFUZZ, to avoid scaring researchers away. As an alternative, projects with such a successful history of community integration can choose to release unprotected binaries to a set of trusted security researchers. On the other hand, the vast majority of software gets far less to no attention. These less well-known pieces of software are still used by many users and they might profit significantly from raising the bar against fuzzing (e.g., industrial controllers such as PLCs [6, 37] or other types of proprietary software).

Furthermore, in this paper, we consider the case that the attacker can use any state-of-the-art bug finding tool. However, we assume that she spends no time on manually reverse engineering the binary or building custom tooling. We are aware that in a more realistic scenario, the target application might be attacked by a human analyst. However, we assume that ANTIFUZZ is combined with other techniques that were developed to incur significant cost for human analyst during reverse engineering [16, 17, 21, 24, 41, 43, 53]. Therefore, to ensure that different concerns (defending against fuzzing *and* defending against analysis by a human) are separated, we explicitly exclude human analysts from our attacker model.

## 4.1 Attacking Coverage-guidance

As mentioned previously, the core assumption of coverage-guided fuzzers is that new coverage indicates new behavior in the program. To undermine this assumption, we modify the program which we want to defend against fuzzing by adding irrelevant code in such a way that its coverage information drowns out the actual signal. More specifically, by adding irrelevant code regions (which we call *fake code*), we deliberately disturb the code coverage tracking mechanisms within fuzzers. Thereby, we weaken the fuzzer's ability to use the feedback mechanism in any useful way and thus remove their advantage over blind fuzzers.

To introduce noise into the coverage information, we use two different techniques. The first technique aims at producing different "interesting" coverage for nearly all inputs. The rationale behind this is that according to the coverage-guide assumption, *any* new coverage means that the fuzzer found an input that causes new behavior. Therefore, if the program *always* displays new coverage (due to our fake code), the fuzzer cannot distinguish between legitimate new coverage and invalid fake coverage. As every single input seems to trigger new behavior, the fuzzer assumes that every input is interesting. Therefore, it spends a significant amount of time on generating mutations based on invalid input.

To implement this technique, we calculate the hash of the program input and based on this hash, we pick a small random subset of fake functions to call. Each fake function recursively calls the next fake function from a table of function pointers, in such a way that we introduce a large number of new edges in the protected program.

Since even a single bit flip in the input causes the hash to be completely different, nearly any input that the fuzzer generates displays new behavior. Fuzzers that are objective-driven and thus assign weights to more interesting code construct might find it easy to distinguish between this simple fake code and the actual application code. Since we cannot assume that future fuzzers will treat new coverage information in the same way as current fuzzers do, we introduce a second technique that aims at providing plausible-looking, semi-hard constraints. The second technique is designed to add fake code that looks like it belongs to the legitimate input handling

code of the original application. At the same time, this code should include a significant number of easy constraints as well as some very hard constraints. These hard constraints can draw the attention of different solving strategies, while the easy constraints allow us to add noise to the true coverage information. We create this fake code by creating random trees of nested conditions with conditions on the input ranging from simple to complicated.

**Evasion** Overall, the attack on the code-coverage assumption consists of a combination of these two techniques to fool the fuzzer into believing that most inputs lead to new code coverage and thus they are classified as "interesting". This fills up the attention mechanism of the fuzzer (e.g., AFL's bitmap or a queue) with random information which breaks the assumption that the feedback mechanism is helpful in determining which inputs will lead to interesting code.

## 4.2 Preventing Crash Detection

After applying our previous method, coverage-guided fuzzers are "blinded" and have few advantages left in comparison to blind fuzzers. To further reduce the ability of both coverage-guided and blind fuzzers to find bugs, we introduced two additional techniques that attack assumption B identified earlier.

There are multiple ways for a fuzzer to detect if a crash has happened. The three most common ways are (i) observing the exit status, (ii) catching the crashing signal by overwriting the signal handler, and (iii) using the operating system (OS) level debugging interfaces such as `ptrace`. To harden our protected program against fuzzers, we try to block these approaches by common anti-debugging measures as well as a custom signal handler that exits the application gracefully. After we install our custom signal handler, we intentionally trigger a segfault (*fake crash*) that our own signal handler recognizes and ignores. This way, if an outside entity is observing crashes that we try to mask, it will always observe a crash for each and every input. It is worth mentioning that by design, the fake crash is triggered at *every* program execution independent from the user input. Thus we do not introduce crashes based on user inputs.

**Evasion** We try to catch all crashes before they are reported to an outside entity. If the current application is under observation or analysis (i.e., where catching crashes is not allowed), the application is terminated. Typically, if it was deemed necessary to apply AntiFuzz to any application, there is likely no scenario where it would *also* be necessary to continue operating under the given conditions.

In all of these cases, no crashes will be detected even if they still occur, which breaks the assumption that a crashing input is detectable as such.

## 4.3 Delaying Execution

We found that fuzzing tools need many executions per second to operate efficiently. Our third countermeasure attacks this assumption, without reducing the overall performance of the protected program, as follows: we check whether the input is a well-formed input; if and only if we detect a malformed input, we enforce an artificial slowdown of the application. For most applications, this would not induce any slowdowns in real-world scenarios, where input files are typically well-formed. But at the same time, it would significantly reduce the execution speed for fuzzers, where most of the inputs will be incorrect. We believe that even if malformed input files occasionally happen in real scenarios, a slowdown of e.g., 250ms per invalid input is barely noticeable to the end user in most cases. In contrast, even such a small delay has drastic effects on fuzzing. Thus, only fuzzers are negatively affected by this technique.

Delaying the execution can happen through different means, the easiest way to cause a delay is using the `sleep()` function. However, to harden this technique against automated code analysis and patching tools, one can add a computationally-heavy task (e.g., encryption, hash calculation, or even cryptocurrency mining) to the protected program such that the resulting solution is necessary to continue the execution.

**Evasion** Most applications expect some kind of structure for their input files and have the ability to tell if the input adheres to this structure. Therefore, AntiFuzz does not need to rely on any formal specification; instead, our responses are triggered by existing error paths within the program. For the prototype implementation, we do not propose to detect error paths automatically, but instead insert them manually as a developer. If the input is malformed, we artificially slow down the execution speed of the program. This breaks the assumption that the application can be executed hundreds or thousands of times per second, thus severely limiting the chances of efficiently finding new code coverage.

## 4.4 Overloading Symbolic Execution Engines

To prevent program analysis techniques from extracting information to solve constraints and cover more code, we introduce two techniques. Both techniques are based on the idea that simple tasks can be rewritten in a way that it is a lot harder to reason about their behavior [51]. For example, we can replace an addition operation using an additive homomorphic encryption scheme. In the following, we introduce two practical techniques to achieve this goal.

First, we use hash comparisons. The idea is to replace all comparisons of input data to constants (e.g., magic bytes) with a comparison of their respective strong cryptographic hash values. While still practically equivalent (unless small collisions for current hashes are found), the resulting computation is significantly more complex. The resulting symbolic

expressions grow significantly, and the solvers fail to find a satisfying assignment for these equations; they become useless for finding correct inputs. However this technique has one weakness: If a seed file is provided that contains the correct value, a concolic execution engine might still be able to continue solving other branches.

As a second technique, we can encrypt and then decrypted the input with a block cipher. We later describe this technique in detail in Section 5.4.

**Evasion** By sending the input data through a strong block cipher and replacing direct comparisons of input data to magic bytes by hash operations, symbolic, concolic, and taint-based execution engines are significantly slowed down and hampered in their abilities to construct valid inputs. This breaks the assumption that constraints in the application are solvable. Even though the encryption/decryption combination is an identity transformation, it is very hard to prove automatically that the resulting output byte only depends on the corresponding input byte. Therefore, symbolic/concolic execution engines either carry very large expressions for each input byte, or they concretize every input byte, completely voiding the advantage they provide. Finally, common taint tracking engines will not be able to infer taint on the input, as the encryption thoroughly mixes the input bits.

# 5 Implementation Details

In this section, we provide an overview of the proof-of-concept implementation of our techniques in a tool called AntiFuzz. As explained above, the use case for AntiFuzz is a developer who has access to source code and wants to protect his application from attackers who use automatic bug finding tools to find bugs cost-effectively. Hence, an important objective was to keep the required modifications to the project at a minimum, so that AntiFuzz is easy to apply. The implementation consists of a Python script that automatically generates a single C header file that needs to be included in the target program. Furthermore, small changes need to be performed to instrumt a given application. For our experiments, we analyzed the time it took us to apply AntiFuzz to LAVA-M (which consists of the four programs `base64`, `md5sum`, `uniq`, and `who`). As we were already familiar with the code base of these tools, we could more closely resemble a developer who has a good understanding of the structure of the code. It took us four to ten minutes to apply AntiFuzz to each application. The number of lines that needed to be added or changed depends on the number of constant comparisons that need to be replaced by hash comparisons. `base64` was an outlier with 79 changed lines, 64 of which were necessary due to a check against every possible character in the `base64` alphabet. The three remaining applications required 6 (`uniq`), 7 (`who`), and 23 (`md5sum`) changed lines, respectively.

In the following, we describe technical details of how AntiFuzz is implemented.

## 5.1 Attacking Coverage-guidance

To prevent coverage-guided fuzzing, it is necessary to generate random constraints, edges, and constant comparisons, as detailed in Section 4.1. The core idea here is to use every byte of the input file in a way that could lead to a new basic block, e.g., by making it depend on some constraints or by comparing it to randomly generated constants. Depending on the configurable number of constraints and the size of the input file, every byte could be part of *multiple* constraints and constant comparisons.

Implementation-wise, although it is possible to generate code for AntiFuzz dynamically at runtime, this might cause problems for fuzzers relying on static code instrumentation (i.e., they might not be able to "see" code introduced by AntiFuzz). Thus, our template engine, implemented in 300 lines of Python code, generates a C file containing all randomly chosen constraints and constants, and further provides the ability to set configuration values (e.g., number of fake basic blocks).

The random edge generation is implemented through a shuffled array (where the input file seeds the randomness) consisting of functions that call each other based on their position in the array (up to a certain configurable depth).

AntiFuzz provides a function called `antifuzz_init()` that needs to be called with the input filename, ideally before the file is being processed by the application. This change needs to be done manually by the developer when he wants to protect his software against fuzzing: the developer needs to add one line that calls this function. The function implements all the techniques against coverage-guided fuzzers mentioned earlier and sets up signal handlers to prevent crash detection, as detailed in the next section.

## 5.2 Preventing Crash Detection

When `antifuzz_init()` is called, AntiFuzz has to confirm that no crashes can be observed. As detailed in Section 4.2, it is necessary to overwrite the crash signal handlers, as well as prevent it from being observed with `ptrace`.

In the former case, AntiFuzz first checks whether overwriting signals is possible: we register a custom signal handler and deliberately crash the application. If the custom signal handler was called, it ignores the crash and resumes execution. If the application does not survive the crash, it means that overwriting signals is not possible and, for our purposes, the resulting crash is a desirable side-effect. If the application survives the crash, evidently, signal overwriting is possible.

AntiFuzz then installs custom signal handlers for all common crash signals and overwrites these with either a timeout or a graceful exit (depending on the configuration). This will

keep some fuzzers from covering any code because they do not survive the artificial crash at the beginning of the application. This behavior could also be replaced by an exit or by calling additional functions that lead to fake code coverage to keep up a facade of a working fuzzer.

In the case of `ptrace`, we use a well-known anti-debugging technique [34] to detect if we are being observed by `ptrace`: we check whether we can `ptrace` our own process. If we can `ptrace` our own process, it means that no other process is `ptrace`ing it. However, if we are unable to `ptrace` our own process, it implies that another process is `ptrace`ing it and therefore ANTIFUZZ terminates the application.

### 5.3 Delaying Execution

As detailed in Section 4.3, ANTIFUZZ needs to know when an input is malformed to slow down the application and hamper the performance of fuzzers. The main idea, implementation-wise, is to allow the developer to inform ANTIFUZZ whenever an input is malformed. Most applications already have some kind of error handling for malformed input, which either discards the input or terminates the application. Within this error handling function of the to-be-protected program, the developer needs to add a single call to `antifuzz_onerror()`.

Upon invocation of `antifuzz_onerror()`, ANTIFUZZ delays the execution for a configurable amount of time using either of the mechanisms mentioned in Section 4.3.

### 5.4 Overloading Symbolic Execution Engines

There are two main parts to our countermeasures against symbolic/concolic execution and taint analysis engines: replacing constant comparisons with comparisons of their respective cryptographic hashes, and putting the input through a cryptographic block cipher before usage.

The first part is implemented via the SHA-512 hash function. The developer needs to replace important (i.e., input-based) comparisons with the hash functions provided by ANTIFUZZ. Due to the nature of cryptographic hashes, two hash values can only be checked for equality, and not whether one is larger or smaller than the other.

To encrypt and decrypt the input buffer, we use the AES-256 encryption function in ECB mode. The key is generated from a hash of the input at runtime. We provide a function that provides the encryption-decryption routine. We can use this function on any kind of input stream. We provide `antifuzz_fread()` as a convenience to make it easier to integrate the common cases. Any call to `fread()` needs to be replaced with its ANTIFUZZ-equivalent call.

Figure 2 illustrates the implementation of all described techniques using ANTIFUZZ in a simple program. Figure 2.a shows an unprotected application which is checking an input value. If the input is valid, it might lead to a program crash caused by a bug. Otherwise, the program will print some error and exit. Figure 2.b illustrates the same program which is now protected by ANTIFUZZ. Additional layers of fake edges and constraints are specifically targeting coverage-guided fuzzers. Further down the control-flow graph of the protected application, ANTIFUZZ added its input encryption/decryption routine. Next in the Figure 2.b, ANTIFUZZ installs its custom signal handler and then causes an intentional segmentation fault (fake crash). However, since ANTIFUZZ installed a custom signal handler, it receives the signal and checks whether it is the fake crash or not. If it is legitimate, it delays the execution and then exits gracefully. This step basically is the anti-crash detection implementation of ANTIFUZZ, which works together with an execution delay mechanism. Finally, in Figure 2.b, we harden the comparison against 1337 with a comparison of hashed values.

## 6 Evaluation

Our evaluation aims to answer the following five research questions (**RQs**):

- **RQ 1.** Are current obfuscation techniques efficient against automated bug-finding via fuzzing?
- **RQ 2.** Are the techniques we designed effective at disrupting the targeted fuzzing assumptions?
- **RQ 3.** Are the techniques effective at preventing fuzzers from finding bugs?
- **RQ 4.** Are the techniques effective at reducing the amount of code that is being tested?
- **RQ 5.** Do our techniques introduce any significant performance overhead?

To answer the first research question **RQ 1.**, we demonstrate that modifying a custom dummy application (which is illustrated in Listing 2) using the state-of-the-art obfuscation tool TIGRESS [15] does not yield a satisfying level of protection against current fuzzers.

Following the answer to **RQ 1.**, we test all our techniques individually on multiple fuzzers to demonstrate that they are effective if and only if the fuzzer employs the targeted assumptions. From this experiment, we can answer **RQ 2.** and conclude that our mitigations are working as intended. We use the same dummy application used in **RQ 1.** to evaluate eight fuzzers and bug-finding tools, namely: AFL 2.52b, VUZZER, HONGGFUZZ 1.6, DRILLER commit 66a3428, ZZUF 0.15, PEACH 3.1.124, and QSYM commit d4bf407. Besides the aforementioned fuzzers, we consider one purely symbolic execution based tool to complete the set of automatic bug finding techniques: KLEE 1.4.0.0 [12].

To answer **RQ 3.**, we test a subset of these fuzzers against the LAVA-M dataset to demonstrate that ANTIFUZZ is able to prevent bug finding in real-world applications.

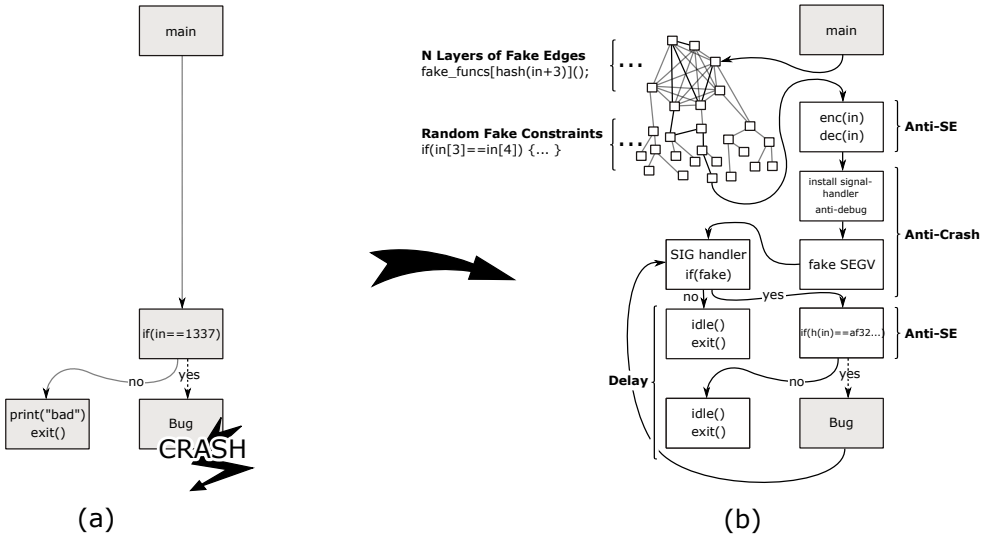



Figure 2: A simple program before (a) and after integration (b) with ANTIFUZZ.

To address **RQ 4.**, we evaluate ANTIFUZZ on binary executables from binutils to show the difference in test coverage in a protected and unprotected application. This experiment demonstrates that ANTIFUZZ does not simply hide bugs, but also drastically reduces the attack surface. It is worth mentioning that in all experiments mentioned above, the bug finding tools were able to find the bugs in a matter of minutes prior to enforcing ANTIFUZZ protection. After applying our techniques, there were *zero* bugs found by the tested tools within a period of 24 hours.

In the last step, we measure the overhead introduced by ANTIFUZZ using the SPEC CPU2006 benchmarking suite to answer **RQ 5.**.

Note that, due to the configurable nature of ANTIFUZZ, we use the following configuration for all experiments:

- **Attacking Coverage-guidance**: Generates 10,000 fake functions with constraints, and 10,000 basic blocks for random edge generation.
- **Delaying Execution**: The signal handler introduces a slowdown in case of a crash to timeout the application (in addition to slowdowns due to malformed inputs). The duration of the sleep is set to 750ms.
- **Preventing Crash Detection**: We enabled all techniques mentioned in Section 5.2.
- **Overloading Symbolic Execution Engines**: Important comparisons for equivalence were replaced with SHA-512 hash comparisons and the input data was encrypted and decrypted via AES-256 in ECB mode.

If the fuzzer supported both binary instrumentation and compile-time instrumentation, we used the compile-time instrumentation. While in reality, a fuzzer would have to use binary-only instrumentation mechanisms (given our attacker model), we chose to use compile-time instrumentation as it achieves better performance and is also more robust. Therefore, we erred on the side of caution by assuming that an attacker is more powerful than state-of-the-art tools.

## 6.1 ANTIFUZZ versus Software Obfuscation

One of the goals of software obfuscation is to prevent security researchers, who rely on traditional *manual* reverse engineering techniques, from finding bugs. In this section, we demonstrate that obfuscation on its own fails to thwart *automatic* bug finding tools.

Intuitively, blind fuzzers without feedback mechanisms are not hindered by obfuscation at all, because they neither have nor need any knowledge about the code. Feedback-driven fuzzers, however, *do* need access to edges and basic blocks to obtain coverage information they can use to guide the fuzzing process. Thus, obfuscating the control flow via common techniques such as control flow flattening or virtual machine based obfuscation [21] might impact coverage-guided fuzzers.

**Experiment** To demonstrate that obfuscation techniques alone do not protect an application from automatic bug finding tools, we obfuscated a dummy application (see Listing 2) with TIGRESS 2.2 [15] and let different fuzzers find the correct crashing input.

Listing 2: Dummy application that crashes if input is 'crsh'

```
int check(char* input, int size) {
  if(size == 4 && input[0] == 'c' && input[1] == 'r' &&
       input[2] == 's' && input[3] == 'h') {
    crash();
  }
}
```

For this experiment, we use AFL, HONGGFUZZ, KLEE and ZZUF which are representative of all three fuzzer categories. Note that VUZZER was excluded because (1) VUZZER is based on the IDA Pro disassembler, which is thwarted by obfuscation before the fuzzing process even begins, and (2) Tigress had trouble compiling non-64bit executables while VUZZER (at the time of the experiment) was not working on 64-bit binaries. Additionally, any fuzzer which is based on the aforementioned tools was excluded from the experiment. For example, QSYM and DRILLER use AFL with an additional symbolic execution engine. Therefore, if AFL is able to find the bug, we conclude that other tools that use AFL under the hood can also find the bug.

We configured TIGRESS by enabling as many of the obfuscation features as we could. The exact configuration is shown in Table 1 of Appendix A.

**Result** This experiment revealed that all fuzzers could find the crashing input despite all obfuscation techniques being enabled. This answers research question **RQ 1.**, current obfuscation techniques are not efficient against automated bug finding techniques. Even though changing the control-flow graph might have an impact on the feedback mechanism, the changes are static or random. In contrast, in ANTIFUZZ the additional information for the feedback mechanism is dependant on the input, which is a major difference between common obfuscation methods and our approach.

## 6.2 Finding Crashes in a Simple Dummy Application

To answer research question **RQ 2.**, we use the same dummy application from the previous experiment.

For this evaluation, we enable our anti-fuzzing techniques one at a time, rather than enabling all of them at once. This allows us to observe which fuzzer is vulnerable to each technique we introduced. We use this rather simple target for two reasons. (1) If a fuzzer is unable to find this very shallow bug, they will most likely also fail to find more complex crashes, and (2) the code is simple enough to be adjusted to different systems and fuzzers (e.g., DRILLER needs CGC binaries).

Any input that is not the crashing input is deemed to be malformed, i.e., ANTIFUZZ decides to slow down the application in that case. If countermeasures against program analysis techniques are activated, the data from the input file is first encrypted and then decrypted again. The comparisons against the individual bytes of "`crsh`" are done via hash comparisons (e.g., `hash("c") == hash(input[0])`). Signal tampering and anti-coverage techniques are all applied before the input file is opened. Since both PEACH and ZZUF are not able to overcome the four-byte constraints on their own, we provided ZZUF with the seed file where only the "c" character was missing. Similarly, PEACH was evaluated on an ELF64 parser (readelf). We modified the elf parser to include an additional one-byte check of a field in ELF64 that guards the crash.

Table 2: Evaluation against the dummy application. ✓ means ANTIFUZZ was successful in preventing bug finding (no crash was found) and ✗means that at least one crashing input was found. `None` means ANTIFUZZ was disabled, `All` means that all techniques against fuzzers (Coverage, Crash, Speed and Symbolic Execution) were turned on.

| | None | Coverage | Crash | Speed | Symbolic Exec. | All |
|---|---|---|---|---|---|---|
| AFL | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ |
| Honggfuzz | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ |
| Vuzzer | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Driller | ✗ | ✓ | - | - | ✗ | ✓ |
| Klee | ✗ | ✓ | ✓ | ✓ | ✓ | ✓[a] |
| zzuff | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ |
| Peach | ✗ | ✗ | ✓ | ✓ | ✗ | ✓ |
| QSYM | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ |

[a] Klee ran at least 24h and then crashed due to memory constraints.

Every possible combination of fuzzer and ANTIFUZZ configuration ran for a period of 24 hours. Moreover, in this experiment, the configuration with all fuzzing countermeasures enabled ("All") ran for a total of 100 hours.

**Result** The results of this experiment are shown in Table 2. Without ANTIFUZZ, it only took a couple of seconds up to a few minutes for all eight fuzzers to find the crashing input. However, when ANTIFUZZ was fully activated, *no* fuzzer was able to do so even after 100 hours. Comparing this table to Table 1 shows that our techniques clearly address the fundamental assumptions that fuzzers use to find bugs.

All coverage-guided fuzzers were impeded by our anti-coverage feature. As expected, all fuzzers were unable to find crashes when we used our anti-crash detection technique. It is worth mentioning that DRILLER was not tested with this configuration because the CGC environment does not allow custom signal handlers. Surprisingly, KLEE was also unable to find the crash because of its incomplete handling of custom signals. Since delaying execution technique (speed) also relies on custom signals, the experiment with DRILLER was omitted and KLEE failed to find the bug. ZZUF was able to crash the target because there were only 256 different inputs to try.

As expected, KLEE was not able to find the correct input once countermeasures against symbolic execution were activated. Surprisingly, VUZZER is confused by this technique as well. A closer inspection suggests that this behavior was due to the fact that this technique is also highly effective at obfuscating taint information.

## 6.3 Finding Crashes in LAVA-M

The dummy application demonstrated our ability to thwart fuzzers for simple examples. To make sure that our techniques also hold up on more complex applications (and answer **RQ 3.**), we evaluate ANTIFUZZ with the LAVA-M dataset [18], which consists of four applications (`base64`, `who`, `uniq` and `md5sum`) where several bugs were artificially

Table 3: Statistical analysis of the code coverage on eight binaries from binutils. The effect size is given in percentage of the branches that could be covered after enabling ANTIFUZZ as compared to the coverage achieved on an unprotected program. Experiments where the two-tailed Mann-Whitney U test resulted in $p < 0.05$ are displayed in bold.

| | addr2line | ar | nm-new | objdump | readelf | size | strings | strip-new |
|---|---|---|---|---|---|---|---|---|
| **vuzzer** | **12.12%, p: 0.04** | - | **1.81%, p: 0.04** | **2.65%, p: 0.03** | **1.10%, p: 0.04** | 13.41%, p: 0.33 | **6.25%, p: 0.04** | 0.84%, p: 0.19 |
| **afl** | **9.49%, p: 0.04** | - | **1.92%, p: 0.04** | **4.98%, p: 0.04** | **0.70%, p: 0.04** | **6.30%, p: 0.04** | **16.17%, p: 0.04** | **4.52%, p: 0.04** |
| **hongg** | **0.00%, p: 0.03** | 0.00%, p: 0.25 | **0.00%, p: 0.03** | **0.00%, p: 0.03** | **0.00%, p: 0.03** | **0.00%, p: 0.03** | **0.00%, p: 0.03** | **0.00%, p: 0.03** |
| **qsym** | **7.12%, p: 0.04** | **11.69%, p: 0.03** | **5.30%, p: 0.04** | **5.47%, p: 0.04** | **1.75%, p: 0.04** | **9.79%, p: 0.04** | **8.55%, p: 0.04** | **4.89%, p: 0.04** |

Table 4: Evaluation against `base64`, `uniq`, `who`, `md5sum` from the LAVA-M data set. ✓ means ANTIFUZZ was successful in preventing bug finding (no crash was found) and ✗ means that at least one crashing input was found, the # sign denotes the number of unique crashes found. `None` means ANTIFUZZ was disabled, `All` means that all techniques against fuzzers (Coverage, Crash, Speed and Symbolic Execution) are turned on.

| | None | Coverage | Crash | Speed | Symbolic Execution | All |
|---|---|---|---|---|---|---|
| **base64** | | | | | | |
| AFL | ✗(#28) | ✓ | ✓ | ✓ | ✗(#24) | ✓ |
| Honggfuzz | ✗(#48) | ✓ | ✓ | ✓ | ✗(#48) | ✓ |
| QSYM | ✗(#48) | ✓ | ✓ | ✓ | ✓ | ✓ |
| Vuzzer | ✗(#47) | ✓ | ✓ | ✓ | ✗(#33) | ✓ |
| zzuf | ✗(#1) | ✗(#1) | ✓ | ✓ | ✗(#1) | ✓ |
| **uniq** | | | | | | |
| AFL | ✗(#14) | ✓ | ✓ | ✓ | ✗(#13) | ✓ |
| Honggfuzz | ✗(#29) | ✓ | ✓ | ✓ | ✗(#29) | ✓ |
| QSYM | ✗(#14) | ✓ | ✓ | ✓ | ✓ | ✓ |
| Vuzzer | ✗(#26) | ✓ | ✓ | ✓ | ✗(#15) | ✓ |
| zzuf | ✗(#1) | ✗(#1) | ✓ | ✓ | ✗(#1) | ✓ |
| **who** | | | | | | |
| AFL | ✗(#194) | ✓ | ✓ | ✓ | ✗(#95) | ✓ |
| Honggfuzz | ✗(#72) | ✓ | ✓ | ✓ | ✗(#72) | ✓ |
| QSYM | ✗(#1926) | ✓ | ✓ | ✓ | ✓ | ✓ |
| Vuzzer | ✗(#266) | ✓ | ✓ | ✓ | ✗(#260) | ✓ |
| zzuf | ✗(#1) | ✗(#1) | ✓ | ✓ | ✗(#1) | ✓ |
| **md5sum** | | | | | | |
| AFL | - | - | - | - | - | - |
| Honggfuzz | ✗(#57) | ✓ | ✓ | ✓ | ✗(#55) | ✓ |
| QSYM | ✗(#34) | ✓ | ✓ | ✓ | ✓ | ✓ |
| Vuzzer | ✗(#25) | ✓ | ✓ | ✓ | ✗(#22) | ✓ |
| zzuf | - | - | - | - | - | - |

inserted. All fuzzer configurations were allowed to run for 24 hours each. Due to `DWORD` comparisons that AFL has difficulty to solve, the AFL modification LAF-INTEL was used, which breaks comparisons (including string operations) down to single byte comparisons to allow for more nuanced edge generation during compilation. For blind fuzzers like ZZUF, solving four bytes is too hard, thus one constraint was reduced to a single bit-flip for this fuzzer alone.

**Results** Table 4 shows our result. The # sign denotes the number of unique crashes found (according to distinct LAVA-M fault IDs). Again we can see the same consistent result for all binaries: once ANTIFUZZ is turned on, it effectively prevents fuzzers from detecting bugs. The exceptional cases are similar to the ones we discussed in the previous section. In summary, these results demonstrate that our anti-fuzzing features are applicable to real-world binaries to prevent bug finding.

## 6.4 Reducing Code Coverage

As a next step, we want to answer **RQ 4.** by demonstrating that applying ANTIFUZZ results in far less coverage in coverage-based fuzzers. More specifically, we evaluated AFL, HONGGFUZZ, VUZZER, and QSYM against eight real-world binaries from the binutils collection (namely `addr2line`, `ar`, `size`, `strings`, `objdump`, `readelf`, `nm-new`, `strip-new`). Every fuzzer and every application was executed three times for 24 hours in the setting "None" (ANTIFUZZ is disabled) and then again in the setting "All" (all ANTIFUZZ features are enabled).

**Result** The results of this experiment are shown in Figure 3. For each of the eight binutils programs, we compare the performance of the four tested fuzzers (measured in the number of branches covered) without and with protection via ANTIFUZZ. It is apparent that ANTIFUZZ does indeed severely hinder fuzzers from extending code coverage. Note that in all cases, when ANTIFUZZ was activated, even after 24 hours the fuzzers could only reach coverage that would have been reached in the first few minutes without ANTIFUZZ.

We performed a statistical analysis on the resulting data, the results are shown in Table 3. All but three out of thirty experiments were statistically significant with $p < 0.05$ according to a two-tailed Mann–Whitney U test. Two of the insignificant results are from VUZZER, which displayed rather low coverage scores even without ANTIFUZZ enabled. The other insignificant result is on `ar`, a target where most bug finding tools fail due to a multi-byte comparison. Additionally, we calculated the reduction of the amount of covered code that resulted from enabling ANTIFUZZ. Typically (in half of the experiments), less than 3% of the code that was tested on an unprotected target could be covered when ANTIFUZZ was enabled. The 95th percentile of coverage was less than 13% of the code that the fuzzers found when targeting an unprotected program. In the worst result, we achieved a reduction to 17%. Therefore, we conclude that ANTIFUZZ will typically reduce the test coverage achieved by 90% to 95%.

## 6.5 Performance Overhead

Lastly, to answer **RQ 5.** we measure the performance overhead caused by using ANTIFUZZ on complex programs. For this purpose, we use the SPEC CPU2006 version 1.1 INT benchmark. This experiment consists of all benchmarks that

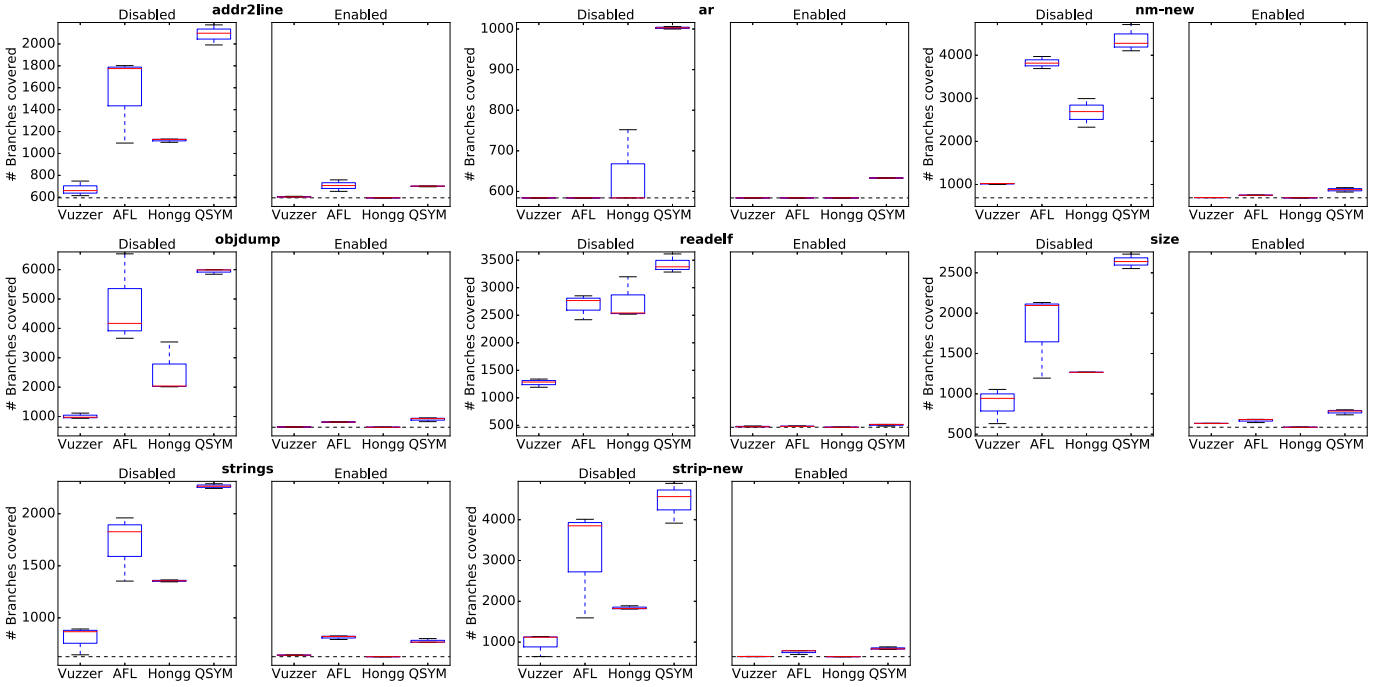



Figure 3: Evaluation of eight binutils binaries to show the branch coverage difference between unmodified binary ("Disabled") and binary with ANTIFUZZ ("Enabled"). The dashed line at the bottom is the baseline (i.e. the number of branches covered with the seed file).

take an input file (thus only 462.libquantum was excluded). The remaining benchmarks ran for three iterations each and were averaged over ten runs with the geometric mean.

**Result** The impact of ANTIFUZZ for each benchmark was insignificant enough to bear little to no observable overhead (see Table 5): most applications show small *negative* overheads (with the outlier being gcc with -3.80%), but the positive overheads also never reach 1%. The total average overhead is -0.42%. This is expected because `antifuzz_init()` is only called once when the input file is opened. Reading the file to memory and checking if the input data is well-formed usually happens only once in the beginning, thus it does not impact the computationally intensive main part of the benchmarks at all.

Table 5: SPEC CPU2006 INT benchmark.

| Benchmark | Overhead |
|---|---|
| 400.perlbench | 0.13% |
| 401.bzip2 | 0.11% |
| 403.gcc | -3.80% |
| 429.mcf | -0.36% |
| 445.gobmk | 0.89% |
| 456.hmmer | 0.32% |
| 458.sjeng | 0.43% |
| 464.h264ref | -1.53% |
| 471.omnetpp | -0.8% |
| 473.astar | -1.06% |
| 483.xalancbmk | 0.17% |
| **Total average** | **-0.42%** |

## 7 Limitations

In the following, we discuss limitations of both our proposed approach and implementation, and also consider threats to validity. For our current prototype implementation, a human analyst can likely remove the protection mechanisms added by ANTIFUZZ rather easily. However, according to our attacker model, we regard this threat out of scope in the context of this paper. Moreover, many other works have detailed techniques to prevent modification and human analysis using software obfuscation techniques [16, 17, 21, 24, 41, 43, 53]. For a more complete protection, we recommend to use a combination of both ANTIFUZZ as well as traditional anti-analysis/-patching techniques.

The delay-inducing technique should not be applied to any kind of public-facing server software, as this would drastically weaken the server against Denial-of-Service attacks. Instead of sleeping or busy waiting, one should implement a similar approach based on rate limitation.

The number of functions added as fake code results in a fixed file size increase of approximately 25MB. While this is less relevant for large software binaries, it might pose significant code size overhead for small binaries. However, for modern machines we deem this to be a minor obstacle.

Furthermore, it is worth mentioning that one can avoid this increase in file size by using self-modifying code. We explicitly decided not to use self-modifying code since such techniques have the tendency of making exploitation easier by using memory pages with read/write/execute privileges and potentially raising alerts in anti-virus products.

Furthermore, ANTIFUZZ in its current form requires developer involvement which is not optimal from a usability perspective. However, most of the manual work in ANTIFUZZ can be automated. In particular, we require the developer to perform the following tasks: (a) find error paths, (b) replace constant comparisons, and (c) annotate functions which read user input or data. It is relatively easy to automate items (b) and (c) via a compiler pass. The reason that finding error paths is more challenging is that there are many different ways for handling errors. On the other hand, the responsible developer is well aware of the error handling code. Adding a single function call in the error handler is straightforward and does not significantly increase the complexity of the code base.

Additionally, it is worth mentioning that the benchmarking suite which we used was focused on CPU intensive tasks rather than I/O bound tasks. We assume that using our prototype AES implementation to encrypt and decrypt every input significantly increases the overhead on I/O bound tasks. Therefore, we recommend to replace AES by a much weaker and faster encryption algorithm, as our goal is not to be cryptographically secure, but to confuse SMT solvers.

Finally, it has to be considered that automatic program transformations for obfuscation can always be thwarted [7]. Therefore, tools like ANTIFUZZ can never completely guarantee that they can defeat a motivated human analyst. Based on this observation, the situation for anti-fuzzing mechanisms like ANTIFUZZ is similar to obfuscation mechanisms: given sufficient interest from the attackers and defenders, a prolonged arms race is to be expected. This also means that as time passes, continuing this arms race will become more and more expensive for both sides involved. However, similar to obfuscation, we expect only the implementation of tools like ANTIFUZZ to become more complicated. Similarly to modern obfuscation tools, usage of anti-fuzzing defenses will most likely remain cheap.

As we cannot evaluate against techniques not yet invented, some of our techniques could be attacked by smarter fuzzers. The junk code that was inserted could be detected based on statistical patterns or the way it interacts with the rest of the execution. To counter this, more complex and individualized junk code fragments could be used. For example, junk code can change global variables that are also used in the original code (e.g., in opaque predicates).

## 8 Related Work

Obfuscating software against program understanding has been exhaustively researched. Common techniques include injecting junk code that is never executed [16, 53], often hidden behind conditional expressions that always evaluate to some fixed value [17]. The control flow can be further cloaked by creating many seemingly dissimilar paths that are picked randomly [43] to thwart dynamic analysis based approaches. Other common techniques include self-modifying code [41], which increases the difficulty of obtaining a useful disassembly and changes to the control-flow [21, 24]. Similarly, there has been some work that specifically target symbolic execution [51].

Recent research tried to address a very similar issue: To increase the cost of the attacker, Hu et al. [35] insert a large number of fake bugs into the target application. This approach has the advantage that it works against many different kinds of attack scenarios. However, they rely on the bugs being non-exploitable as otherwise the actual security of the application is reduced. For example, the authors state that they rely on the exact stack layout behavior of the chosen compiler. Any update to the compiler might render the previously "safe" bugs exploitable. Additionally, fuzzers generally tend to find many hundreds to thousands of crashes for each real bug uncovered. Adding some more crashes does not prevent the fuzzer from finding real bugs. The large number of crashes found might draw attention and common analysis techniques for bug triage (such as AFLs bug exploration mode) will greatly simplify weeding out the fake bugs.

In contrast, our approach is much more low key. Additionally, since in our approach no proper test coverage is achieved, no analysis of the produced fuzzing data will be able to uncover any bugs. An idea similar to our fake code insertions was also presented in a talk by Kang et al. [38]. However, they explicitly tried to prevent AFL in QEMU mode from finding a specific crashing path. In our scenario, the defenders do not know the specific crashing path, as otherwise, they would rather fix the bug. Additionally, as we demonstrated in our evaluation, our approach is effective across different fuzzers and does not attack a specific implementation.

Finally, a master thesis by Göransson and Edholm has introduced the idea of masking crashes and actively detecting if the program is being fuzzed, e.g., by detecting specific AFL environment variables [30]. Similarly to the work by Kang et al., the methods they devised are highly specific to the implementation of the only two fuzzers they considered: AFL and HONGGFUZZ. Additionally, to reduce the execution speed of fuzzers, they proposed to artificially decrease the overall performance of the program under test, whereas ANTIFUZZ only decreases the performance if the input is malformed.

## 9 Conclusion

In this paper, we categorized the general assumptions common to all current bug-finding tools. Based on this analysis, we developed techniques to systematically attack and break these assumptions (and thus a representative sample of contempo-

rary fuzzers). The evaluation demonstrated that obfuscation on its own fails to prevent fuzzing satisfyingly. In contrast, our techniques effectively prevent fuzzers from finding crashing inputs in simple programs, even if the crash was found in seconds in an unprotected application. Furthermore, we demonstrated that we get the same result for real-world applications, i.e., fuzzers are unable to detect any crashes or even achieve a significant amount of new code coverage. Our techniques also show no significant overhead when evaluated with the SPEC benchmark suite and can, therefore, be easily and efficiently integrated into projects with negligible impact to the performance.

In summary, we conclude that the techniques presented in this paper are well applicable to deter automated, dragnet-style hunting for bugs. In combination with common program obfuscation techniques, they will also hinder a targeted attack, as manual work is needed to reverse engineer and remove the anti-fuzzing measures before a more cost-efficient, automated fuzzing campaign can be started.

## Acknowledgments

We would like to thank our shepherd Mathias Payer and the anonymous reviewers for their valuable comments and suggestions. This work was supported by the German Federal Ministry of Education and Research (BMBF Grant 16KIS0592K HWSec) and the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States - EXC 2092 CASA. In addition, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 786669 (ReAct). This paper reflects only the authors' view. The Research Executive Agency is not responsible for any use that may be made of the information it contains.

## References

[1] Announcing oss-fuzz: Continuous fuzzing for open source software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. Accessed: 2019-02-18.

[2] Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/. Accessed: 2019-02-18.

[3] Peach. http://www.peachfuzzer.com/. Accessed: 2019-02-18.

[4] Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: 2019-02-18.

[5] zzuf. https://github.com/samhocevar/zzuf. Accessed: 2019-02-18.

[6] Ali Abbasi, Thorsten Holz, Emmanuele Zambon, and Sandro Etalle. ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers. In *Annual Computer Security Applications Conference (ACSAC)*, 2017.

[7] Andrew W. Appel. Deobfuscation is in NP, 2002.

[8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[9] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[13] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[14] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

[15] Christian Collberg. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/. Accessed: 2019-02-18.

[16] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[17] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[18] Brendan Dolan, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy*, 2016.

[19] Christopher Domas. Movfuscator: Turning 'mov' into a soul-crushing RE nightmare. https://recon.cx/2015/slides/recon2015-14-christopher-domas-The-movfuscator.pdf. Accessed: 2019-02-18.

[20] Joe W. Duran and Simeon Ntafos. A report on random testing. In *International Conference on Software Engineering (ICSE)*, 1981.

[21] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. Multi-stage binary code obfuscation using improved virtual machine. In *International Conference on Information Security (ISC)*, 2011.

[22] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy*, 2018.

[23] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering (ICSE)*, 2009.

[24] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. Control flow based obfuscation. In *ACM Workshop on Digital Rights Management (DRM)*, 2005.

[25] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[27] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[28] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[29] Peter Goodman. Shin GRR: Make Fuzzing Fast Again. https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again/. Accessed: 2019-02-18.

[30] David Göransson and Emil Edholm. Escaping the Fuzz. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016.

[31] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, 2013.

[32] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[33] Aki Helin. A general-purpose fuzzer. https://gitlab.com/akihe/radamsa. Accessed: 2019-02-18.

[34] Thorsten Holz and Frédéric Raynal. Detecting honeypots and other suspicious environments. *IEEE Information Assurance Workshop*, 2005.

[35] Zhenghao Hu, Yu Hu, and Brendan Dolan-Gavitt. Chaff bugs: Deterring attackers by making software buggier.

[36] Richard Johnson. Go speed tracer. https://talos-intelligence-site.s3.amazonaws.com/production/document_files/files/000/000/048/original/Go_Speed_Tracer.pdf. Accessed: 2019-02-18.

[37] Anastasis Keliris and Michail Maniatakos. Icsref: A framework for automated reverse engineering of industrial control systems binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[38] Kang Li, Yue Yin, and Guodong Zhu. Afl's blindspot and how to resist afl fuzzing for arbitrary elf binaries. https://www.blackhat.com/us-18/briefings/schedule/index.html#afls-blindspot-and-how-to-resist-afl-fuzzing-for-arbitrary-elf-binaries-11048. Accessed: 2019-02-18.

[39] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[40] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[41] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *International Workshop on Information Security Applications (WISA)*, 2005.

[42] David Molnar, Xue Cong Li, and David Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, 2009.

[43] Andre Pawlowski, Moritz Contag, and Thorsten Holz. Probfuscation: an obfuscation approach using probabilistic control flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[44] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[45] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[46] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan M Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[47] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *USENIX Security Symposium*, 2017.

[48] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[49] Vincent Ulitzsch, Bhargava Shastry, and Dominik Maier. Follow the white rabbit simplifying fuzz testing using fuzzexmachina. https://i.blackhat.com/us-18/Thu-August-9/us-18-Ulitzsch-Follow-The-White-Rabbit-Simplifying-Fuzz-Testing-Using-FuzzExMachina.pdf/. Accessed: 2019-02-18.

[50] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[51] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear obfuscation to combat symbolic execution. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[52] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[53] Gregory Wroblewski. *General method of program code obfuscation*. PhD thesis, Wroclaw University of Technology, 2002.

[54] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[55] Michael Zalewski. "technical whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2019-02-18.

[56] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2019-02-18.

## A TIGRESS Configuration

Table 1: Tigress configuration for ANTIFUZZ evaluation. Asterisk means: "apply to all functions".

| Transform | Functions |
|---|---|
| Virtualize | check |
| Flatten | * |
| Split | check |
| InitOpaque | main |
| EncodeLiterals | * |
| EncodeArithmetic | * |
| AddOpaque | * |
| AntiTaintAnalysis | * |
| UpdateOpaque | * |
| Ident | * |
| InitEntropy | main |
| AntiAliasAnalysis | * |
| InitBranchFuns | check |
| RandomFuns | * |
| InitImplicitFlow | check |