FUZZIFICATION: Anti-Fuzzing Techniques

Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee[†], Taesoo Kim

Georgia Institute of Technology
[†] *University of Georgia*

Abstract

Fuzzing is a software testing technique that quickly and automatically explores the input space of a program without knowing its internals. Therefore, developers commonly use fuzzing as part of test integration throughout the software development process. Unfortunately, it also means that such a blackbox and the automatic natures of fuzzing are appealing to adversaries who are looking for zero-day vulnerabilities.

To solve this problem, we propose a new *mitigation* approach, called FUZZIFICATION, that helps developers protect the released, binary-only software from attackers who are capable of applying state-of-the-art fuzzing techniques. Given a performance budget, this approach aims to hinder the fuzzing process from adversaries as much as possible. We propose three FUZZIFICATION techniques: 1) SpeedBump, which amplifies the slowdown in normal executions by hundreds of times to the fuzzed execution, 2) BranchTrap, interfering with feedback logic by hiding paths and polluting coverage maps, and 3) AntiHybrid, hindering taint-analysis and symbolic execution. Each technique is designed with best-effort, defensive measures that attempt to hinder adversaries from bypassing FUZZIFICATION.

Our evaluation on popular fuzzers and real-world applications shows that FUZZIFICATION effectively reduces the number of discovered paths by 70.3% and decreases the number of identified crashes by 93.0% from real-world binaries, and decreases the number of detected bugs by 67.5% from LAVA-M dataset while under user-specified overheads for common workloads. We discuss the robustness of FUZZIFICATION techniques against adversarial analysis techniques. We open-source our FUZZIFICATION system to foster future research.

1 Introduction

Fuzzing is a software testing technique that aims to find software bugs automatically. It keeps running the program with randomly generated inputs and waits for bug-exposing behaviors such as crashing or hanging. It has become a standard practice to detect security problems in complex, modern software [40, 72, 37, 25, 23, 18, 9]. Recent research has built several efficient fuzzing tools [57, 52, 29, 34, 6, 64] and found a large number of security vulnerabilities [51, 72, 59, 26, 10].

Unfortunately, advanced fuzzing techniques can also be used by malicious attackers to find zero-day vulnerabilities. Recent studies [61, 58] confirm that attackers predominantly prefer fuzzing tools over others (*e.g.*, reverse engineering) in finding vulnerabilities. For example, a survey of information security experts [28] shows that fuzzing techniques discover 4.83 times more bugs than static analysis or manual detection. Therefore, developers might want to apply *anti-fuzzing* techniques on their products to hinder fuzzing attempts by attackers, similar in concept to using obfuscation techniques to cripple reverse engineering [12, 13].

In this paper, we propose a new direction of binary protection, called FUZZIFICATION, that hinders attackers from effectively finding bugs. Specifically, attackers may still be able to find bugs from the binary protected by FUZZIFICATION, but with significantly more effort (*e.g.*, CPU, memory, and time). Thus, developers or other trusted parties who get the original binary are able to detect program bugs and synthesize patches before attackers widely abuse them. An effective FUZZIFICATION technique should enable the following three features. First, it should be effective for hindering existing fuzzing tools, finding fewer bugs within a fixed time; second, the protected program should still run efficiently in normal usage; third, the protection code should not be easily identified or removed from the protected binary by straightforward analysis techniques.

No existing technique can achieve all three goals simultaneously. First, software obfuscation techniques, which impede static program analysis by randomizing binary representations, seem to be effective in thwarting fuzzing attempts [12, 13]. However, we find that it falls short of FUZZIFICATION in two ways. Obfuscation introduces unacceptable overhead to normal program executions. Figure 1(a) shows that obfuscation slows the execution by at least 1.7 times when using UPX [60] and up to 25.0 times when using

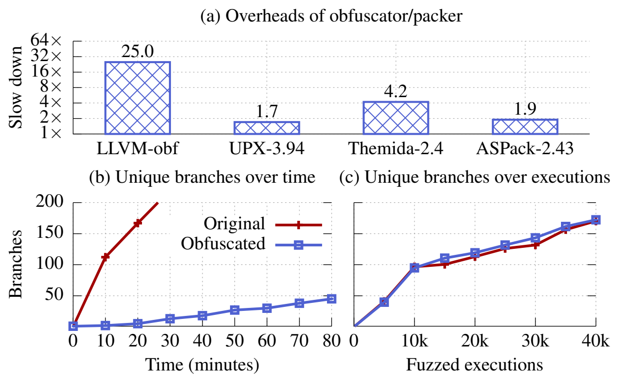

Figure 1: Impact of obfuscation techniques on fuzzing. (a) Obfuscation techniques introduce 1.7×-25.0× execution slow down. (b) and (c) fuzzing obfuscated binaries discovers fewer program paths over time, but gets a similar number of paths over executions.

LLVM-obfuscator [33]. Also, obfuscation cannot effectively hinder fuzzers in terms of path exploration. It can slow each fuzzed execution, as shown in Figure 1(b), but the path discovery per execution is almost identical to that of fuzzing the original binary, as shown in Figure 1(c). Therefore, obfuscation is not an ideal FUZZIFICATION technique. Second, software diversification changes the structure and interfaces of the target application to distribute diversified versions [35, 3, 53, 50]. For example, the technique of N-version software [3] is able to mitigate exploits because attackers often depend on clear knowledge of the program states. However, software diversification is powerless on hiding the original vulnerability from the attacker's analysis; thus it is not a good approach for FUZZIFICATION.

In this paper, we propose three FUZZIFICATION techniques for developers to protect their programs from malicious fuzzing attempts: SpeedBump, BranchTrap, and AntiHybrid. The SpeedBump technique aims to slow program execution during fuzzing. It injects delays to *cold* paths, which normal executions rarely reach but that fuzzed executions frequently visit. The BranchTrap technique inserts a large number of input-sensitive jumps into the program so that any input drift will significantly change the execution path. This will induce coverage-based fuzzing tools to spend their efforts on injected bug-free paths instead of on the real ones. The AntiHybrid technique aims to thwart hybrid fuzzing approaches that incorporate traditional fuzzing methods with dynamic taint analysis and symbolic execution.

We develop defensive mechanisms to hinder attackers identifying or removing our techniques from protected binaries. For SpeedBump, instead of calling the `sleep` function, we inject randomly synthesized CPU-intensive operations to cold paths and create control-flow and data-flow dependencies between the injected code and the original code. We reuse existing binary code to realize BranchTrap to prevent an adversary from identifying the injected branches.

To evaluate our FUZZIFICATION techniques, we apply them on the LAVA-M dataset and nine real-world applications, including `libjpeg`, `libpng`, `libtiff`, `pcre2`, `readelf`, `objdump`, `nm`, `objcopy`, and `MuPDF`. These programs are extensively used to evaluate the effectiveness of fuzzing tools [19, 11, 48, 67]. Then, we use four popular fuzzers —AFL, HonggFuzz, VUzzer, and QSym— to fuzz the original programs and the protected ones for the same amount of time. On average, fuzzers detect 14.2 times more bugs from the original binaries and 3.0 times more bugs from the LAVA-M dataset than those from "fuzzified" ones. At the same time, our FUZZIFICATION techniques decrease the total number of discovered paths by 70.3%, and maintain user-specified overhead budget. This result shows that our FUZZIFICATION techniques successfully decelerate fuzzing performance on vulnerability discovery. We also perform an analysis to show that data-flow and control-flow analysis techniques cannot easily disarm our techniques.

In this paper, we make the following contributions:

- We first shed light on the new research direction of anti-fuzzing schemes, so-called, FUZZIFICATION.
- We develop three FUZZIFICATION techniques to slow each fuzzed execution, to hide path coverage, and to thwart dynamic taint-analysis and symbolic execution.
- We evaluate our techniques on popular fuzzers and common benchmarks. Our results show that the proposed techniques hinder these fuzzers, finding 93% fewer bugs from the real-world binaries and 67.5% fewer bugs from the LAVA-M dataset, and 70.3% less coverage while maintaining the user-specified overhead budget.

We will release the source code of our work at `https://github.com/sslab-gatech/fuzzification`.

2 Background and Problem

2.1 Fuzzing Techniques

The goal of fuzzing is to automatically detect program bugs. For a given program, a fuzzer first creates a large number of inputs, either by random mutation or by format-based generation. Then, it runs the program with these inputs to see whether the execution exposes unexpected behaviors, such as a crash or an incorrect result. Compared to manual analysis or static analysis, fuzzing is able to execute the program orders of magnitude more times and thus can explore more program states to maximize the chance of finding bugs.

2.1.1 Fuzzing with Fast Execution

A straightforward way to improve fuzzing efficiency is to make each execution faster. Current research highlights several fast execution techniques, including (1) customized system and hardware to accelerate fuzzed execution and (2) parallel fuzzing to amortize the absolute execution time in large-scale. Among these techniques, AFL uses the fork server and persistent mode to avoid the heavy process creation and can accelerate fuzzing by a factor of two or more [68, 69]. AFL-PT, kAFL, and HonggFuzz utilize hardware features such as Intel Process Tracing (PT) and Branch Trace Store (BTS) to collect code coverage efficiently to guide fuzzing [65, 54, 23]. Recently, Xu *et al.* designed new operating system primitives, like efficient system calls, to speed up fuzzing on multi-core machines [64].

2.1.2 Fuzzing with Coverage-guidance

Coverage-guided fuzzing collects the code coverage for each fuzzed execution and prioritizes fuzzing the input that has triggered new coverage. This fuzzing strategy is based on two empirical observations: (1) a higher path coverage indicates a higher chance of exposing bugs; and (2) mutating inputs that ever trigger new paths is likely to trigger another new path. Most popular fuzzers take code coverage as guidance, like AFL, HonggFuzz, and LibFuzzer, but with different methods for coverage representation and coverage collection.

Coverage representation. Most fuzzers take basic blocks or branches to represent the code coverage. For example, HonggFuzz and VUzzer use basic block coverage, while AFL instead considers the branch coverage, which provides more information about the program states. Angora [11] combines branch coverage with the call stack to further improve coverage accuracy. However, the choice of representation is a trade-off between coverage accuracy and performance, as more fine-grained coverage introduces higher overhead to each execution and harms the fuzzing efficiency.

Coverage collection. If the source code is available, fuzzers can instrument the target program during compilation or assembly to record coverage at runtime, like in AFL-LLVM mode and LibFuzzer. Otherwise, fuzzers have to utilize either static or dynamic binary instrumentation to achieve a similar purpose, like in AFL-QEMU mode [70]. Also, several fuzzers leverage hardware features to collect the coverage [65, 54, 23]. Fuzzers usually maintain their own data structure to store coverage information. For example, AFL and HonggFuzz use a fixed-size array and VUzzer utilizes a *Set* data structure in Python to store their coverage. However, the size of the structure is also a trade-off between accuracy and performance: an overly small memory cannot capture every coverage change, while an overly large memory introduces significant overhead. For example, AFL's performance drops 30% if the bitmap size is changed from 64KB to 1MB [19].

2.1.3 Fuzzing with Hybrid Approaches

Hybrid approaches are proposed to help solve the limitations of existing fuzzers. First, fuzzers do not distinguish input bytes with different types (*e.g.*, magic number, length specifier) and thus may waste time mutating less important bytes that cannot affect any control flow. In this case, taint analysis is used to help find which input bytes are used to determine branch conditions, like VUzzer [52]. By focusing on the mutation of these bytes, fuzzers can quickly find new execution paths. Second, fuzzers cannot easily resolve complicated conditions, such as comparison with magic value or checksum. Several works [57, 67] utilize symbolic execution to address this problem, which is good at solving complicated constraints but incurs high overhead.

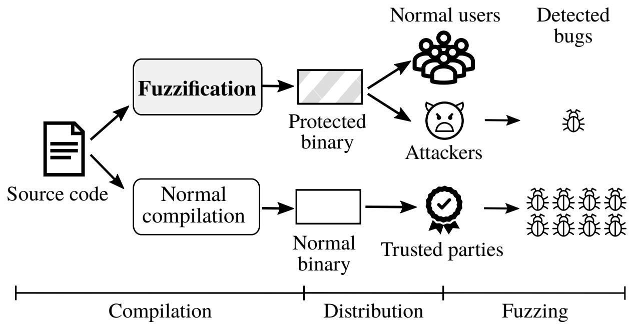

Figure 2: Workflow of FUZZIFICATION protection. Developers create a protected binary with FUZZIFICATION techniques and release it to public. Meanwhile, they send the normally compiled binary to trusted parties. Attackers cannot find many bugs from the protected binary through fuzzing, while trusted parties can effectively find significantly more bugs and developers can patch them in time.

2.2 FUZZIFICATION Problem

Program developers may want to completely control the bug-finding process, as any bug leakage can bring attacks and lead to financial loss [45]. They demand exposing bugs by themselves or by trusted parties, but not by malicious end-users. Anti-fuzzing techniques can help to achieve that by decelerating unexpected fuzzing attempts, especially from malicious attackers.

We show the workflow of FUZZIFICATION in Figure 2. Developers compile their code in two versions. One is compiled with FUZZIFICATION techniques to generate a protected binary, and the other is compiled normally to generate a normal binary. Then, developers distribute the protected binary to the public, including normal users and malicious attackers. Attackers fuzz the protected binary to find bugs. However, with the protection of FUZZIFICATION techniques, they cannot find as many bugs quickly. At the same time, developers distribute the normal binary to trusted parties. The trusted parties can launch fuzzing on the normal binary with the native speed and thus can find more bugs in a timely manner. Therefore, developers who receive bug reports from trusted parties can fix the bug before attackers widely abuse it.

Anti-fuzz candidates	Effective	Generic	Efficient	Robust
Pack & obfuscation	✔	✔	✗	✔
Bug injection	✔	✔	✗	✗
Fuzzer identification	✔	✗	✔	✗
Emulator bugs	✔	✗	✔	✔
FUZZIFICATION	✔	✔	✔	✔

Table 1: Possible design choices and evaluation with our goals.

2.2.1 Threat Model

We consider motivated attackers who attempt to find software vulnerabilities through state-of-the-art fuzzing techniques, but with limited resources like computing power (at most similar resources as trusted parties). Adversaries have the binary protected by FUZZIFICATION and they have knowledge of our FUZZIFICATION techniques. They can use off-the-shelf binary analysis techniques to disarm FUZZIFICATION from the protected binary. Adversaries who have access to the unprotected binary or even to program source code (*e.g.*, inside attackers, or through code leakage) are out of the scope of this study.

2.2.2 Design Goals and Choices

A FUZZIFICATION technique should achieve the following four goals simultaneously:

- **Effective:** It should effectively reduce the number of bugs found in the protected binary, compared to that found in the original binary.
- **Generic:** It tackles the fundamental principles of fuzzing and is generally applicable to most fuzzers.
- **Efficient:** It introduces minor overhead to the normal program execution.
- **Robust:** It is resistant to the adversarial analysis trying to remove it from the protected binary.

With these goals in mind, we examine four design choices for hindering malicious fuzzing, shown in Table 1. Unfortunately, no method can satisfy all goals.

Packing/obfuscation. Software packing and obfuscation are mature techniques against reverse engineering, both generic and robust. However, they usually introduce higher performance overhead to program executions, which not only hinders fuzzing, but also affects the use of normal users.

Bug injection. Injecting arbitrary code snippets that trigger non-exploitable crashes can cause additional bookkeeping overhead and affect end users in unexpected ways [31].

Fuzzer identification. Detecting the fuzzer process and changing the execution behavior accordingly can be bypassed easily (*e.g.*, by changing fuzzer name). Also, we cannot enumerate all fuzzers or fuzzing techniques.

Emulator bugs. Triggering bugs in dynamic instrumentation tools [4, 14, 38] can interrupt fuzzing [42, 43]. However, it requires strong knowledge of the fuzzer, so it is not generic.

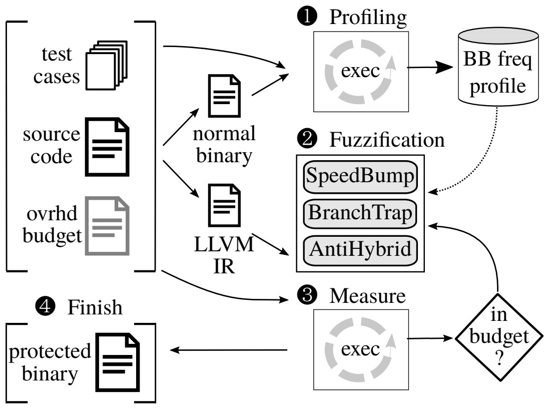

Figure 3: Overview of FUZZIFICATION process. It first runs the program with given test cases to get the execution frequency profile. With the profile, it instruments the program with three techniques. The protected binary is released if it satisfies the overhead budget.

2.3 Design Overview

We propose three FUZZIFICATION techniques – SpeedBump, BranchTrap, and AntiHybrid– to target each fuzzing technique discussed in §2.1. First, SpeedBump injects fine-grained delay primitives into cold paths that fuzzed executions frequently touch but normal executions rarely use (§3). Second, BranchTrap fabricates a number of input-sensitive branches to induce the coverage-based fuzzers to waste their efforts on fruitless paths (§4). Also, it intentionally saturates the code coverage storage with frequent path collisions so that the fuzzer cannot identify interesting inputs that trigger new paths. Third, AntiHybrid transforms explicit data-flows into implicit ones to prevent data-flow tracking through taint analysis, and inserts a large number of spurious symbols to trigger path explosion during the symbolic execution (§5).

Figure 3 shows an overview of our FUZZIFICATION system. It takes the program source code, a set of commonly used test cases, and an overhead budget as input and produces a binary protected by FUZZIFICATION techniques. Note that FUZZIFICATION relies on developers to determine the appropriate overhead budget, whatever they believe will create a balance between the functionality and security of their production. ❶ We compile the program to generate a normal binary and run it with the given normal test cases to collect basic block frequencies. The frequency information tells us which basic blocks are rarely used by normal executions. ❷ Based on the profile, we apply three FUZZIFICATION techniques to the program and generate a temporary protected binary. ❸ We measure the overhead of the temporary binary with the given normal test cases again. If the overhead is over the budget, we go back to step ❷ to reduce the slow down to the program, such as using shorter delay and adding less instrumentation. If the overhead is far below the budget, we increase the overhead accordingly. Otherwise, ❹ we generate the protected binary.

3 SpeedBump: Amplifying Delay in Fuzzing

We propose a technique called SpeedBump to slow the fuzzed execution while minimizing the effect to normal executions. Our observation is that the fuzzed execution frequently falls into paths such as error-handling (*e.g.*, wrong MAGIC bytes) that the normal executions rarely visit. We call them the *cold* paths. Injecting delays in cold paths will significantly slow fuzzed executions but will not affect regular executions that much. We first identify cold paths from normal executions with the given test cases and then inject crafted delays into least-executed code paths. Our tool automatically determines the number of code paths to inject delays and the length of each delay so that the protected binary has overhead under the user-defined budget during normal executions.

Basic block frequency profiling. FUZZIFICATION generates a basic block frequency profile to identify cold paths. The profiling process follows three steps. First, we instrument the target programs to count visited basic blocks during the execution and generate a binary for profiling. Second, with the user-provided test cases, we run this binary and collect the basic blocks visited by each input. Third, FUZZIFICATION analyzes the collected information to identify basic blocks that are rarely executed or never executed by valid inputs. These blocks are treated as cold paths in delay injection.

Our profiling does not require the given test cases to cover 100% of all legitimate paths, but just to trigger the commonly used functionalities. We believe this is a practical assumption, as experienced developers should have a set of test cases covering most of the functionalities (*e.g.*, regression test-suites). Optionally, if developers can provide a set of test cases that trigger uncommon features, our profiling results will be more accurate. For example, for applications parsing well-known file formats (*e.g.*, `readelf` parses ELF binaries), collecting valid/invalid dataset is straightforward.

Configurable delay injection. We perform the following two steps repeatedly to determine the set of code blocks to inject delays and the length of each delay:

- We start by injecting a 30ms delay to 3% of the least-executed basic blocks in the test executions. We find that this setting is close enough to the final evaluation result.
- We measure the overhead of the generated binary. If it does not exceed the user-defined overhead budget, we go to the previous step to inject more delay into more basic blocks. Otherwise, we use the delay in the previous round as the final result.

Our SpeedBump technique is especially useful for developers who generally have a good understanding of their applications, as well as the requirements for FUZZIFICATION. We provide five options that developers can use to finely tune SpeedBump's effect. Specifically, `MAX_OVERHEAD` defines the overhead budget. Developers can specify any value as long as they feel comfortable with the overhead. `DELAY_LENGTH` specifies the range of delays. We use 10ms to 300ms in the evaluation. `INCLUDE_INCORRECT` determines whether or not to inject delays to error-handling basic blocks (*i.e.*, locations that are *only* executed by invalid inputs), which is enabled by default. `INCLUDE_NON_EXEC` and `NON_EXEC_RATIO` specify whether to inject delays into how ever many basic blocks are never executed during test execution. This is useful when developers do not have a large set of test cases.

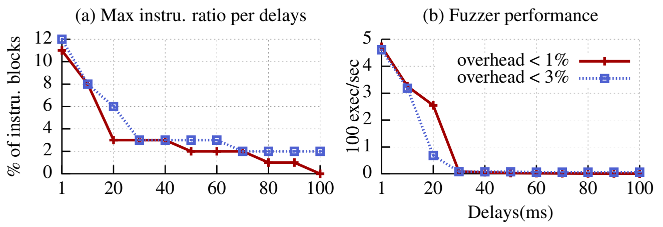

Figure 4: Protecting `readelf` with different overhead budgets. While satisfying the overhead budget, (a) demonstrates the maximum ratio of instrumentation for each delay length, and (b) displays the execution speed of AFL-QEMU on protected binaries.

Figure 4 demonstrates the impact of different options on protecting the `readelf` binary with SpeedBump. We collect 1,948 ELF files on the Debian system as valid test cases and use 600 text and image files as invalid inputs. Figure 4(a) shows the maximum ratio of basic blocks that we can inject delay into while introducing overhead less than 1% and 3%. For a 1ms delay, we can instrument 11% of the least-executed basic blocks for a 1% overhead budget and 12% for 3% overhead. For a 120ms delay, we cannot inject any blocks for 1% overhead and can inject only 2% of the cold paths for 3% overhead. Figure 4(b) shows the actual performance of AFL-QEMU when it fuzzes SpeedBump-protected binaries. The ratio of injected blocks is determined as in Figure 4(a). The result shows that SpeedBump with a 30ms delay slows the fuzzer by more than 50×. Therefore, we use 30ms and the corresponding 3% instrumentation as the starting point.

3.1 Analysis-resistant Delay Primitives

As attackers may use program analysis to identify and remove simple delay primitives (*e.g.*, calling `sleep`), we design robust primitives that involve arithmetic operations and are connected with the original code base. Our primitives are based on CSmith [66], which can generate random and bug-free code snippets with refined options. For example, CSmith can generate a function that takes parameters, performs arithmetic operations, and returns a specific type of value. We modified CSmith to generate code that has data dependencies and code dependencies to the original code. Specifically, we pass a variable from the original code to the generated code as an argument, make a reference from the generated code to the original one, and use the return value to modify a global variable of the original code. Figure 5 shows an example of our delay primitives. It declares a local variable `PASS_VAR`

```
1  //Predefined global variables
2  int32_t GLOBAL_VAR1 = 1, GLOBAL_VAR2 = 2;
3  //Randomly generated code
4  int32_t * func(int32_t p6) {
5     int32_t *l0[1000];
6     GLOBAL_VAR1 = 0x4507L; // affect global var.
7     int32_t *l1 = &g8[1][0];
8     for (int i = 0; i < 1000; i++)
9        l0[i] = p6;          // affect local var from argv.
10    (*g7) = func2(g6++);
11    (*g5) |= ~(!func3(**g4 = ~0UL));
12    return l1;              // affect global var.
13 }
14 //Inject above function for delay
15 int32_t PASS_VAR = 20;
16 GLOBAL_VAR2 = func(PASS_VAR);
```

Figure 5: Example delay primitive. Function `func` updates global variables to build data-flow dependency with original program.

and modifies global variables `GLOBAL_VAR1` and `GLOBAL_VAR2`. In this way, we introduce data-flow dependency between the original code and the injected code (line 6, 9 and 12), and change the program state without affecting the original program. Although the code is randomly generated, it is tightly coupled with the original code via data-flow and control-flow dependencies. Therefore, it is non-trivial for common binary analysis techniques, like dead-code elimination, to distinguish it from the original code. We repeatedly run the modified CSmith to find appropriate code snippets that take a specific time (*e.g.*, 10ms) for delay injection.

Safety of delay primitives. We utilize the safety checks from CSmith and FUZZIFICATION to guarantee that the generated code is bug-free. First, we use CSmith's default safety checks, which embed a collection of tests in the code, including integer, type, pointer, effect, array, initialization, and global variable. For example, CSmith conducts pointer analysis to detect any access to an out-of-scope stack variable or null pointer dereference, uses explicit initialization to prevent uninitialized usage, applies math wrapper to prevent unexpected integer overflow, and analyzes qualifiers to avoid any mismatch. Second, FUZZIFICATION also has a separate step to help detect bad side effects (*e.g.*, crashes) in delay primitives. Specifically, we run the code 10 times with fixed arguments and discard it if the execution shows any error. Finally, FUZZIFICATION embeds the generated primitives with the same fixed argument to avoid errors.

Fuzzers aware of error-handling blocks. Recent fuzzing proposals, like VUzzer [52] and T-Fuzz [48], identify error-handling basic blocks through profiling and exclude them from the code coverage calculation to avoid repetitive executions. This may affect the effectiveness of our SpeedBump technique, which uses a similar profiling step to identify cold paths. Fortunately, the cold paths from SpeedBump include not only error-handling basic blocks, but also rarely executed functional blocks. Further, we use similar methods to identify error-handling blocks from the cold paths and provide developers the option to choose not to instrument these blocks. Thus, our FUZZIFICATION will focus on instrumenting rarely executed functional blocks to maximize its effectiveness.

4 BranchTrap: Blocking Coverage Feedback

Code coverage information is widely used by fuzzers to find and prioritize interesting inputs [72, 37, 23]. We can make these fuzzers *diligent fools* if we insert a large number of conditional branches whose conditions are sensitive to slight input changes. When the fuzzing process falls into these branch traps, coverage-based fuzzers will waste their resources to explore (a huge number of) worthless paths. Therefore, we propose the technique of BranchTrap to deceive coverage-based fuzzers by misleading or blocking the coverage feedback.

4.1 Fabricating Fake Paths on User Input

The first method of BranchTrap is to *fabricate* a large number of conditional branches and indirect jumps, and inject them into the original program. Each fabricated conditional branch relies on some input bytes to determine to take the branch or not, while indirect jumps calculate their targets based on user input. Thus, the program will take different execution paths even when the input slightly changes. Once a fuzzed execution triggers the fabricated branch, the fuzzer will set a higher priority to mutate that input, resulting in the detection of more fake paths. In this way, the fuzzer will keep wasting its resources (*i.e.*, CPU and memory) to inspect fruitless but bug-free fake paths.

To effectively induce the fuzzers focusing on fake branches, we consider the following four design aspects. First, BranchTrap should fabricate a sufficient number of fake paths to affect the fuzzing policy. Since the fuzzer generates various variants from one interesting input, fake paths should provide different coverage and be directly affected by the input so that the fuzzer will keep unearthing the trap. Second, the injected new paths introduce minimal overhead to regular executions. Third, the paths in BranchTrap should be deterministic regarding user input, which means that the same input should go through the same path. The reason is that some fuzzers can detect and ignore non-deterministic paths (*e.g.*, AFL ignores one input if two executions with it take different paths). Finally, BranchTrap cannot be easily identified or removed by adversaries.

A trivial implementation of BranchTrap is to inject a jump table and use some input bytes as the index to access the table (*i.e.*, different input values result in different jump targets). However, this approach can be easily nullified by simple adversarial analysis. We design and implement a robust BranchTrap with code-reuse techniques, similar in concept to the well-known return-oriented programming (ROP) [55].

4.1.1 BranchTrap with CFG Distortion

To harden BranchTrap, we diversify the return addresses of each injected branch according to the user input. Our idea is inspired by ROP, which reuses existing code for malicious at-

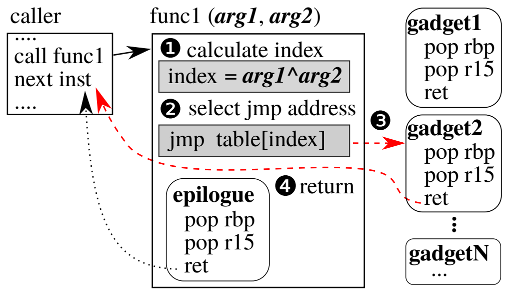

Figure 6: BranchTrap by reusing the existing ROP gadgets in the original binary. Among functionally equivalent gadgets, BranchTrap picks the one based on function arguments.

tacks by chaining various small code snippets. Our approach can heavily distort the program control-flow and makes nullifying BranchTrap more challenging for adversaries. The implementation follows three steps. First, BranchTrap collects function epilogues from the program assembly (generated during program compilation). Second, function epilogues with the same instruction sequence are grouped into one jump table. Third, we rewrite the assembly so that the function will retrieve one of several equivalent epilogues from the corresponding jump table to realize the original function return, using some input bytes as the jump table index. As we replace the function epilogue with a functional equivalent, it guarantees the identical operations as the original program.

Figure 6 depicts the internal of the BranchTrap implementation at runtime. For one function, BranchTrap ❶ calculates the `XOR`ed value of all arguments. BranchTrap uses this value for indexing the jump table (*i.e.*, candidates for epilogue address). ❷ BranchTrap uses this value as the index to visit the jump table and obtains the concrete address of the epilogue. To avoid out-of-bounds array access, BranchTrap divides the `XOR`ed value by the length of the jump table and takes the remainder as the index. ❸ After determining the target jump address, the control-flow is transferred to the gadget (*e.g.*, the same `pop rbp; pop r15; ret` gadget). ❹ Finally, the execution returns to the original return address.

The ROP-based BranchTrap has three benefits:

- **Effective:** Control-flow is constantly and sensitively changed together with the user input mutation; thus FUZZIFICATION can introduce a sufficient number of unproductive paths and make coverage feedback less effective. Also, BranchTrap guarantees the same control-flow on the same input (*i.e.*, deterministic path) so that the fuzzer will not ignore these fake paths.
- **Low overhead:** BranchTrap introduces low overhead to normal user operations (*e.g.*, less than 1% overhead) due to its lightweight operations (Store argument; XOR; Resolve jump address; Jump to gadget).
- **Robust:** The ROP-based design significantly increases the complexity for an adversary to identify or patch the binary. We evaluate the robustness of BranchTrap against adversarial analysis in §6.4.

4.2 Saturating Fuzzing State

The second method of BranchTrap is to saturate the *fuzzing state*, which blocks the fuzzers from learning the progress in the code coverage. Different from the first method, which induces fuzzers focusing on fruitless inputs, our goal here is to prevent the fuzzers from finding real interesting ones. To achieve this, BranchTrap inserts a massive number of branches to the program, and exploits the coverage representation mechanism of each fuzzer to mask new findings. BranchTrap is able to introduce an extensive number (*e.g.*, 10K to 100K) of deterministic branches to some rarely visited basic blocks. Once the fuzzer reaches these basic blocks, its coverage table will quickly fill up. In this way, most of the newly discovered paths in the following executions will be treated as *visited*, and thus the fuzzer will discard the input that in fact explores interesting paths. For example, AFL maintains a fixed-size bitmap (*i.e.*, 64KB) to track edge coverage. By inserting a large number of distinct branches, we significantly increase the probability of bitmap collision and thus reduce the coverage inaccuracy.

Figure 7(a) demonstrates the impact of bitmap saturation on fuzzing `readelf`. Apparently, a more saturated bitmap leads to fewer path discoveries. Starting from an empty bitmap, AFL identifies over 1200 paths after 10 hours of fuzzing. For the 40% saturation rate, it only finds around 950 paths. If the initial bitmap is highly filled, such as 80% saturation, AFL detects only 700 paths with the same fuzzing effort.

Fuzzers with collision mitigation. Recent fuzzers, like CollAFL [19], propose to mitigate the coverage collision issue by assigning a unique identifier to each path coverage (*i.e.*, branch in case of CollAFL). However, we argue that these techniques will not effectively undermine the strength of our BranchTrap technique on saturating coverage storage for two reasons. First, current collision mitigation techniques require program source code to assign unique identifiers during the linking time optimization [19]. In our threat model, attackers cannot obtain the program source code or the original binary – they only have a copy of the protected binary, which makes it significantly more challenging to apply similar ID-assignment algorithms. Second, these fuzzers still have to adopt a fixed size storage of coverage because of the overhead of large storage. Therefore, if we can saturate 90% of the storage, CollAFL can only utilize the remaining 10% for ID-assignment; thus the fuzzing performance will be significantly affected.

4.3 Design Factors of BranchTrap

We provide developers an interface to configure ROP-based BranchTrap and coverage saturation for optimal protection. First, the number of generated fake paths of ROP-based BranchTrap is configurable. BranchTrap depends on the number of functions to make a distorted control-flow. Therefore, injected BranchTrap is effective when the original program

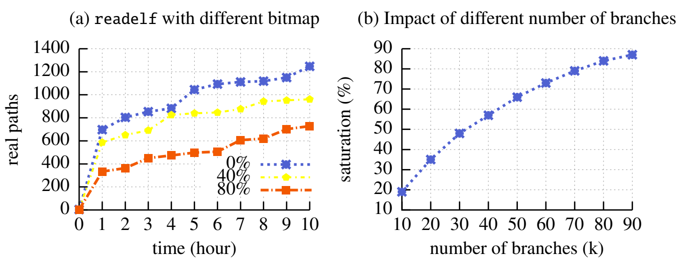

Figure 7: (a) AFL performance with different initial bitmap saturation. (b) Impact on bitmap with different number of branches.

contains plenty of functions. For binaries with fewer functions, we provide an option for developers to split existing basic blocks into multiple ones, each connected with conditional branches. Second, the size of the injected branches for saturating the coverage is also controllable. Figure 7(b) shows how the bitmap can be saturated in AFL by increasing the branch number. It clearly shows that more branches can fill up more bitmap entries. For example, 100K branches can fill up more than 90% of a bitmap entry. Injecting a massive number of branches into the program increases the output binary size. When we inject 100k branches, the size of the protected binary is 4.6MB larger than the original binary. To avoid high code size overhead, we inject a huge number of branches into only one or two of the most rarely executed basic blocks. As long as one fuzzed execution reaches such branches, the coverage storage will be filled and the following fuzzing will find fewer interesting inputs.

5 AntiHybrid: Thwarting Hybrid Fuzzers

A hybrid fuzzing method utilizes either symbolic execution or dynamic taint analysis to improve fuzzing efficiency. Symbolic (or concolic) execution is good at solving complicated branch conditions (*e.g.*, magic number and checksum), and therefore can help fuzzers bypass these hard-to-mutate roadblocks. DTA (Dynamic Taint Analysis) helps find input bytes that are related to branch conditions. Recently, several hybrid fuzzing methods have been proposed and successfully discovered security-critical bugs. For example, Driller [57] adapted selective symbolic execution and proved its efficacy during the DARPA Cyber Grand Challenge (CGC). VUzzer [52] utilized dynamic taint analysis to identify path-critical input bytes for effective input mutation. QSym [67] suggested a fast concolic execution technique that can be scalable on real-world applications.

Nevertheless, hybrid approaches have well-known weaknesses. First, both symbolic execution and taint analysis consume a large amount of resources such as CPU and memory, limiting them to analyzing simple programs. Second, symbolic execution is limited by the path explosion problem. If complex operation is required for processing symbols, the symbolic execution engine has to exhaustively explore and evaluate all execution states; then, most of the symbolic execution engines fail to run to the end of the execution path. Third, DTA analysis has difficulty in tracking implicit data dependencies, such as covert channels, control channels, or timing-based channels. For example, to cover data dependency through a control channel, the DTA engine has to aggressively propagate the taint attribute to any variable after a conditional branch, making the analysis more expensive and the result less accurate.

```
1  char input[] = ...; /* user input */
2  int  value   = ...; /* user input */
3
4  // 1. using implicit data-flow to copy input to antistr
5  //    original code: if (!strcmp(input, "condition!")) { ... }
6  char antistr[strlen(input)];
7  for (int i = 0; i<strlen(input); i++){
8    int ch = 0, temp = 0, temp2 = 0;
9    for (int j = 0; j<8; j++){
10     temp = input[i];
11     temp2 = temp & (1<<j);
12     if (temp2 != 0) ch |= 1<<j;
13   }
14   antistr[i] = ch;
15 }
16 if (!strcmp(antistr, "condition!")) { ... }
17
18 // 2. exploding path constraints
19 //    original code: if (value == 12345)
20 if (CRC_LOOP(value) == OUTPUT_CRC) { ... }
```

Figure 8: Example of AntiHybrid techniques. We use implicit data-flow (line 6-15) to copy strings to hinder dynamic taint analysis. We inject hash function around equal comparison (line 20) to cripple symbolic execution engine.

Introducing implicit data-flow dependencies. We transform the explicit data-flows in the original program into implicit data-flows to hinder taint analysis. FUZZIFICATION first identifies branch conditions and interesting information sinks (*e.g.*, strcmp) and then injects data-flow transformation code according to the variable type. Figure 8 shows an example application of AntiHybrid, where array `input` is used to decide branch condition and `strcmp` is an interesting sink function. Therefore, FUZZIFICATION uses implicit data-flows to copy the array (line 6-15) and replaces the original variable to the new one (line 16). Due to the transformed implicit data-flow, the DTA technique cannot identify the correct input bytes that affect the branch condition at line 16.

Implicit data-flow hinders data-flow analysis that tracks direct data propagation. However, it cannot prevent data dependency inference through differential analysis. For example, recent work, RedQueen [2], infers the potential relationship between input and branch conditions through pattern matching, and thus can bypass the implicit data-flow transformation. However, RedQueen requires the branch condition value to be explicitly shown in the input, which can be easily fooled through simple data modification (*e.g.*, adding the same constant value to both operands of the comparison).

Exploding path constraints. To hinder hybrid fuzzers using symbolic execution, FUZZIFICATION injects multiple code chunks to intentionally trigger path explosions. Specifi-

Project	Version	Program	Arg.	Seeds	Overhead (Binary size)				Overhead (CPU)			
					Speed	BranchTrap	AntiHybrid	All	Speed	BranchTrap	AntiHybrid	All
libjpeg	2017.7	djpeg		GIT	9.0% (0.1M)	101.5% (1.2M)	0.3% (0.0M)	103.2% (1.3M)	1.5%	0.9%	0.3%	2.4%
libpng	1.6.27	readpng		GIT	6.2% (0.1M)	56.0% (1.3M)	0.9% (0.0M)	65.7% (1.5M)	1.8%	2.0%	0.3%	4.0%
libtiff	4.0.6	tiffinfo		GIT	9.2% (0.2M)	72.5% (1.5M)	0.8% (0.0M)	77.3% (1.6M)	1.0%	2.1%	0.5%	4.8%
pcre2	10	pcre2test		built-in	12.9% (0.2M)	85.3% (1.3M)	0.8% (0.0M)	108.6% (1.7M)	1.2%	1.2%	1.0%	3.1%
binutils	2.23	readelf	-a	ELF files	9.6% (0.2M)	77.3% (1.3M)	0.2% (0.0M)	81.0% (1.4M)	1.0%	0.9%	0.9%	3.1%
		objdump	-d		1.4% (0.1M)	17.0% (1.3M)	0.1% (0.0M)	17.5% (1.3M)	1.6%	2.0%	0.9%	4.6%
		nm			1.9% (0.1M)	23.1% (1.2M)	0.1% (0.0M)	23.3% (1.2M)	1.8%	1.6%	1.1%	4.5%
		objcopy	-S		1.7% (0.1M)	20.2% (1.3M)	0.1% (0.0M)	20.6% (1.3M)	1.7%	0.8%	0.5%	2.9%
Average					6.5%	56.6%	0.4%	62.1%	1.4%	1.4%	0.7%	3.7%

Table 2: Code size overhead and performance overhead of fuzzified binaries. `GIT` means Google Image Test-suite. We set performance overhead budget as 5%. For size overhead, we show the percentage and the increased size.

cally, we replace each comparison instruction by comparing the hash values of the original comparison operands. We adopt the hash function because symbolic execution cannot easily determine the original operand with the given hash value. As hash functions usually introduce non-negligible overhead to program execution, we utilize the lightweight cyclic redundancy checking (CRC) loop iteration to transform the branch condition to reduce performance overhead. Although theoretically CRC is not as strong as hash functions for hindering symbolic execution, it also introduces significant slow down. Figure 8 shows an example of the path explosion instrumentation. To be specific, FUZZIFICATION changes the original condition `(value == 12345)` to `(CRC_LOOP(value) == OUTPUT_CRC)` (at line 20). If symbolic execution decides to solve the constraint of the CRC, it will mostly return a timeout error due to the complicated mathematics. For example, QSym, a state-of-the-art fast symbolic execution engine, is armed with many heuristics to scale on real-world applications. When QSym first tries to solve the complicated constraint that we injected, it will fail due to the timeout or path explosion. Once injected codes are run by the fuzzer multiple times, QSym identifies the repetitive basic blocks (*i.e.*, injected hash function) and performs *basic block pruning*, which decides not to generate a further constraint from it to assign resources into a new constraint. After that, QSym will not explore the condition with the injected hash function; thus, the code in the branch can be explored rarely.

6 Evaluation

We evaluate our FUZZIFICATION techniques to understand their effectiveness on hindering fuzzers from exploring program code paths (§6.1) and detecting bugs (§6.2), their practicality of protecting real-world large programs (§6.3), and their robustness against adversarial analysis techniques (§6.4).

Implementation. Our FUZZIFICATION framework is implemented in a total of 6,559 lines of Python code and 758 lines of C++ code. We implement the SpeedBump technique as an LLVM pass and use it to inject delays into cold blocks during the compilation. For the BranchTrap, we analyze the assembly code and modify it directly. For the AntiHybrid technique, we use an LLVM pass to introduce the path explosion and utilize a python script to automatically inject implicit data-flows. Currently, our system supports all three FUZZIFICATION techniques on 64bit applications, and is able to protect 32bit applications except for the ROP-based BranchTrap.

Tasks	Target	AFL	HonggFuzz	QSym	VUzzer
Coverage	8 binaries	O,S,B,H,A	O,S,B,H,A	O,S,B,H,A	–
	MuPDF	O,A	O,A	O,A	–
Crash	4 binaries	O,A	O,A	O,A	–
	LAVA-M	O,A	O,A	O,A	O,A

Table 3: Experiments summary. Protection options: **O**riginal, **S**peedBump, **B**ranchTrap, Anti**H**ybrid, **A**ll. We use 4 binutils binaries, 4 binaries from Google OSS project and `MuPDF` to measure the code coverage. We use binutils binaries and LAVA-M programs to measure the number of unique crashes.

Experimental setup. We evaluate FUZZIFICATION against four state-of-the-art fuzzers that work on binaries, specifically, AFL in QEMU mode, HonggFuzz in Intel-PT mode, VUzzer 32[1], and QSym with AFL-QEMU. We set up the evaluation on two machines, one with Intel Xeon CPU E7-8890 v4@2.20GHz, 192 processors and 504 GB of RAM, and another with Intel Xeon CPU E7-4820@2.00GHz, 32 processors and 128 GB of RAM.

To get reproducible results, we tried to eliminate the non-deterministic factors from fuzzers: we disable the address space layout randomization of the experiment machine and force the deterministic mode for AFL. However, we have to leave the randomness in HonggFuzz and VUzzer, as they do not support deterministic fuzzing. Second, we used the same set of test cases for basic block profiling in FUZZIFICATION, and fed the same seed inputs for different fuzzers. Third,

[1]We also tried to use VUzzer64 to fuzz different programs, but it did not find any crashes even for any original binary after three-day fuzzing. Since VUzzer64 is still experimental, we will try the stable version in the future.

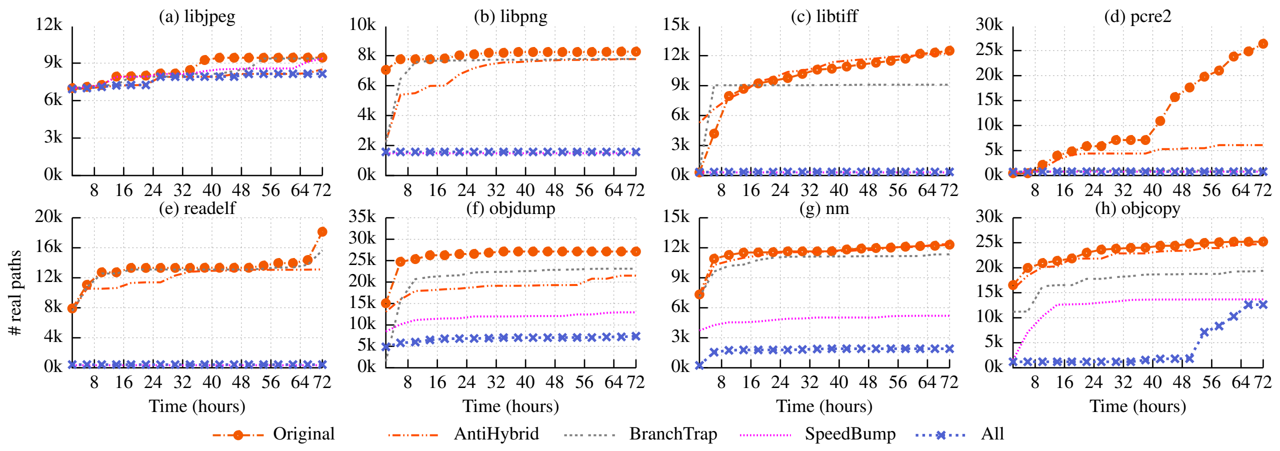

Figure 9: Paths discovered by AFL-QEMU from real-world programs. Each program is compiled with five settings: original (no protection), SpeedBump, BranchTrap, AntiHybrid, and all protections. We fuzz them with AFL-QEMU for three days.

Category	Option	Design Choice
SpeedBump	max_overhead	2%
	delay_length	10ms to 300ms
	include_invalid	True
	include_non_exec	True (5%)
BranchTrap	max_overhead	2%
	bitmap_saturation	40% of 64k bitmap
AntiHybrid	max_overhead	1%
	include_non_exec	True (5%)
Overall	max_overhead	5%

Table 4: Our configuration values for the evaluation.

we used identical FUZZIFICATION techniques and configurations when we conducted code instrumentation and binary rewriting for each target application. Last, we pre-generated FUZZIFICATION primitives (*e.g.*, SpeedBump codes for 10ms to 300ms and BranchTrap codes with deterministic branches), and used the primitives for all protections. Note that developers should use different primitives for the actual releasing binary to avoid code pattern matching analysis.

Target applications. We select the LAVA-M data set [17] and nine real-world applications as the fuzzing targets, which are commonly used to evaluate the performance of fuzzers [11, 19, 64, 52]. The nine real-world programs include four applications from the Google fuzzer test-suite [24], four programs from the binutils [20] (shown in Table 2), and the PDF reader `MuPDF`. We perform two sets of experiments on these binaries, summarized in Table 3. First, we fuzz nine real-world programs with three fuzzers (all except VUzzer[2]) to measure the impact of FUZZIFICATION on finding code paths. Specifically, we compile eight real-world programs (all except `MuPDF`) with five different settings: original (no protection), SpeedBump, BranchTrap, AntiHybrid, and a combination of three techniques (full protection). We compile `MuPDF` with two settings for simplicity: no protection and full protection. Second, we use three fuzzers to fuzz four binutils programs and all four fuzzers to fuzz LAVA-M programs to evaluate the impact of FUZZIFICATION on unique bug finding. All fuzzed programs in this step are compiled in two versions: with no protection and with full protection. We compiled the LAVA-M program to a 32bit version in order to be comparable with previous research. Table 4 shows the configuration of each technique used in our compilation. We changed the fuzzer's timeout if the binaries cannot start with the default timeout (*e.g.*, 1000 ms for AFL-QEMU).

[2]Due to time limit, we only use VUzzer 32 to finding bugs from LAVA-M programs. We plan to do other evaluations in the future.

Evaluation metric. We use two metrics to measure the effectiveness of FUZZIFICATION: code coverage in terms of discovered *real path*s, and unique crashes. Real path is the execution path shown in the original program, excluding the fake ones introduced by BranchTrap. We further excluded the real paths triggered by seed inputs so that we can focus on the ones discovered by fuzzers. Unique crash is measured as the input that can make the program crash with a distinct real path. We filter out duplicate crashes that are defined in AFL [71] and are widely used by other fuzzers [11, 36].

6.1 Reducing Code Coverage

6.1.1 Impact on Normal Fuzzers

We measure the impact of FUZZIFICATION on reducing the number of real paths against AFL-QEMU and HonggFuzz-Intel-PT. Figure 9 shows the 72-hour fuzzing result from AFL-QEMU on different programs with five protection settings. The result of HonggFuzz-Intel-PT is similar and we leave it in Appendix A.

In summary, with all three techniques, FUZZIFICATION can reduce discovered real paths by 76% to AFL, and by

	SpeedBump	BranchTrap	AntiHybrid	All
AFL-QEMU	-66%	-23%	-18%	-74%
HonggFuzz (PT)	-44%	-14%	-7%	-61%
QSym (AFL-QEMU)	-59%	-58%	-67%	-80%
Average	-56%	-31%	-30%	-71%

Table 5: Reduction of discovered paths by FUZZIFICATION techniques. Each value is an average of the fuzzing result from eight real-world programs, as shown in Figure 9 and Figure 10.

67% to HonggFuzz, on average. For AFL, the reduction rate varies from 14% to 97% and FUZZIFICATION reduces over 90% of path discovery for `libtiff`, `pcre2` and `readelf`. For HonggFuzz, the reduction rate is between 38% to 90% and FUZZIFICATION only reduces more than 90% of paths for `pcre2`. As FUZZIFICATION automatically determines the details for each protection to satisfy the overhead budget, its effect varies for different programs.

Table 5 shows the effect of each technique on hindering path discovery. Among them, SpeedBump achieves the best protection against normal fuzzers, followed by BranchTrap and AntiHybrid. Interestingly, although AntiHybrid is developed to hinder hybrid approaches, it also helps reduce the discovered paths in normal fuzzers. We believe this is mainly caused by the slow down in fuzzed executions.

We measured the overhead by different FUZZIFICATION techniques, on program size and execution speed. The result is given in Table 2. In summary, FUZZIFICATION satisfies the user-specified overhead budget, but shows relatively high space overhead. On average, binaries armed with FUZZIFICATION are 62.1% larger than the original ones. The extra code mainly comes from the BranchTrap technique, which inserts massive branches to achieve bitmap saturation. Note that the extra code size is almost the same across different programs. Therefore, the size overhead is high for small programs, but is negligible for large applications. For example, the size overhead is less than 1% for LibreOffice applications, as we show in Table 7. Further, BranchTrap is configurable, and developers may inject a smaller number of fake branches to small programs to avoid large-size overhead.

Analysis on less effective results. FUZZIFICATION shows less effectiveness on protecting the `libjpeg` application. Specifically, it decreases the number of real paths on `libjpeg` by 13% to AFL and by 37% to HonggFuzz, whereas the average reduction is 76% and 67%, respectively. We analyzed FUZZIFICATION on `libjpeg` and find that SpeedBump and BranchTrap cannot effectively protect `libjpeg`. Specifically, these two techniques only inject nine basic blocks within the user-specified overhead budget (2% for SpeedBump and 2% for BranchTrap), which is less than 0.1% of all basic blocks. To address this problem, developers may increase the overhead budget so that FUZZIFICATION can insert more roadblocks to protect the program.

6.1.2 Impact on Hybrid Fuzzers

We also evaluated FUZZIFICATION's impact on code coverage against QSym, a hybrid fuzzer that utilizes symbolic execution to help fuzzing. Figure 10 shows the number of real paths discovered by QSym from the original and protected binaries. Overall, with all three techniques, FUZZIFICATION can reduce the path coverage by 80% to QSym on average, and shows consistent high effectiveness on all tested programs. Specifically, the reduction rate varies between 66% (`objdump`) to 90% (`readelf`). The result of libjpeg shows an interesting pattern: QSym finds a large number of real paths from the original binary in the last 8 hours, but it did not get the same result from any protected binary. Table 5 shows that AntiHybrid achieves the best effect (67% path reduction) against hybrid fuzzers, followed by SpeedBump (59%) and BranchTrap (58%).

Comparison with normal fuzzing result. QSym uses efficient symbolic execution to help find new paths in fuzzing, and therefore it is able to discover 44% more real paths than AFL from original binaries. As we expect, AntiHybrid shows the most impact on QSym (67% reduction), and less effect on AFL (18%) and HonggFuzz (7%). With our FUZZIFICATION techniques, QSym shows less advantage over normal fuzzers, reduced from 44% to 12%.

6.2 Hindering Bug Finding

We measure the number of unique crashes that fuzzers find from the original and protected binaries. Our evaluation first fuzzes four binutils programs and LAVA-M applications with three fuzzers (all but VUzzer). Then we fuzz LAVA-M programs with VUzzer, where we compiled them into 32bit versions and excluded the protection of ROP-based BranchTrap, which is not implemented yet for 32bit programs.

6.2.1 Impact on Real-World Applications

Figure 11 shows the total number of unique crashes discovered by three fuzzers in 72 hours. Overall, FUZZIFICATION reduces the number of discovered crashes by 93%, specifically, by 88% to AFL, by 98% to HonggFuzz, and by 94% to QSym. If we assume a consistent crash-discovery rate along the fuzzing process, fuzzers have to take 40 times more effort to detect the same number of crashes from the protected binaries. As the crash-discovery rate usually reduces over time in real-world fuzzing, fuzzers will have to take much more effort. Therefore, FUZZIFICATION can effectively hinder fuzzers and makes them spend significantly more time discovering the same number of crash-inducing inputs.

6.2.2 Impact on LAVA-M Dataset

Compared with other tested binaries, LAVA-M programs are smaller in size and simpler in operation. If we inject a

	who	uniq	base64	md5sum	Average
Overhead (Size)	17.1% (0.3M)	220.6% (0.3M)	220.0% (0.3M)	210.7% (0.3M)	167.1%
Overhead (CPU)	22.7%	13.2%	21.1%	6.5%	15.9%

Table 6: Overhead of FUZZIFICATION on LAVA-M binaries (all protections except ROP-based BranchTrap) . The overhead is higher as LAVA-M binaries are relatively small (*e.g.*, ≈ 200KB).

Category	**Program**	**Version**	**Overhead**	
			Size	**CPU**
LibreOffice	Writer	6.2	< 1% (+1.3 MB)	0.4%
	Calc		< 1% (+1.3 MB)	0.4%
	Impress		< 1% (+1.3 MB)	0.2%
Music Player	Clementine	1.3	4.3% (+1.3 MB)	0.5%
PDF Reader	MuPDF	1.13	4.1% (+1.3 MB)	2.2%
Image Viewer	Nomacs	3.10	21% (+1.2 MB)	0.7%
Average			5.4%	0.73%

Table 7: FUZZIFICATION on GUI applications. The CPU overhead is calculated on the application launching time. Due to the fixed code injection, code size overhead is negligible for these large applications.

1ms delay on 1% of rarely executed basic block on `who` binary, the program will suffer a slow down of more than 40 times. To apply FUZZIFICATION on the LAVA-M dataset, we allow higher overhead budget and apply more fine-grained FUZZIFICATION. Specifically, we used tiny delay primitives (*i.e.*, 10 µs to 100 µs), tuned the ratio of basic block instrumentation from 1% to 0.1%, reduced the number of applied AntiHybrid components, and injected smaller deterministic branches to reduce the code size overhead. Table 6 shows the run-time and space overhead of the generated LAVA-M programs with FUZZIFICATION techniques.

After fuzzing the protected binaries for 10 hours, AFL-QEMU does not find any crash. HonggFuzz detects three crashes from the original `uniq` binary and cannot find any crash from any protected binary. Figure 12 illustrates the fuzzing result of VUzzer and QSym. Overall, FUZZIFICATION can reduce 56% of discovered bugs to VUzzer and 78% of discovered bugs to QSym. Note that the fuzzing result on the original binaries is different from the ones reported in the original papers [67, 52] for several reasons: VUzzer and QSym cannot eliminate non-deterministic steps during fuzzing; we run the AFL part of each tool in QEMU mode; LAVA-M dataset is updated with several bug fixes[3].

6.3 Anti-fuzzing on Realistic Applications

To understand the practicality of FUZZIFICATION on large and realistic applications, we choose six programs that have a graphical user interface (GUI) and depend on tens of libraries. As fuzzing large and GUI programs is a well-known challenging problem, our evaluation here focuses on measuring the overhead of FUZZIFICATION techniques and the functionality of protected programs. When applying the SpeedBump technique, we have to skip the basic block profiling step due to the lack of command-line interface (CLI) support (*e.g.*, `readelf` parses ELF file and displays results in command line); thus, we only insert slow down primitives into error-handling routines. For the BranchTrap technique, we choose to inject massive fake branches into basic blocks near the entry point. In this way, the program execution will always pass the injected component so that we can measure runtime overhead correctly. We apply the AntiHybrid technique directly.

For each protected application, we first manually run it with multiple inputs, including given test cases, and confirm that FUZZIFICATION does not affect the program's original functionality. For example, `MuPDF` successfully displays, edits, saves, and prints all tested PDF documents. Second, we measure the code size and runtime overhead of the protected binaries for given test cases. As shown in Table 7, on average, FUZZIFICATION introduces 5.4% code size overhead and 0.73% runtime overhead. Note that the code size overhead is much smaller than that of previous programs (*i.e.*, 62.1% for eight relatively small programs Table 2 and over 100% size overhead for simple LAVA-M programs Table 6).

Anti-fuzzing on MuPDF. We also evaluated the effectiveness of FUZZIFICATION on protecting `MuPDF` against three fuzzers – AFL, HonggFuzz, and QSym– as `MuPDF` supports the CLI interface through the tool called "mutool." We compiled the binary with the same parameter shown in Table 4 and performed basic block profiling using the CLI interface. After 72-hours of fuzzing, no fuzzer finds any bug from `MuPDF`. Therefore, we instead compare the number of real paths between the original binary and the protected one. As shown in Figure 13, FUZZIFICATION reduces the total paths by 55% on average, specifically, by 77% to AFL, by 36% to HonggFuzz, and 52% to QSym. Therefore, we believe it is more challenging for real-world fuzzers to find bugs from protected applications.

6.4 Evaluating Best-effort Countermeasures

We evaluate the robustness of FUZZIFICATION techniques against off-the-shelf program analysis techniques that adver-

	Pattern matching	**Control analysis**	**Data analysis**	**Manual analysis**
SpeedBump	✔	✔	✔	-
BranchTrap	✔	✔	✔	-
AntiHybrid	-	✔	✔	-

Table 8: Defense against adversarial analysis. ✔ indicates that the FUZZIFICATION technique is resistant to that adversarial analysis.

[3]https://github.com/panda-re/lava/search?q=bugfix&type=Commits

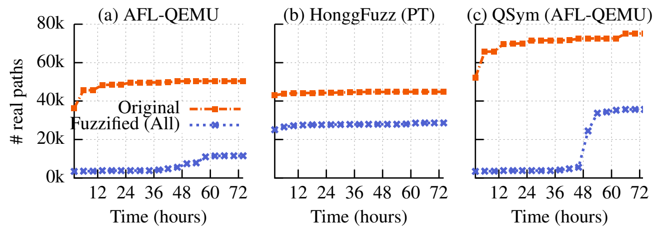

Figure 13: Paths discovered by different fuzzers from the original `MuPDF` and the one protected by three FUZZIFICATION techniques.

saries may use to reverse our protections. However, the experiment results do not particularly indicate that FUZZIFICATION is robust against strong adversaries with incomparable computational resources.

Table 8 shows the analysis we covered and summarizes the evaluation result. First, attackers may search particular code patterns from the protected binary in order to identify injected protection code. To test anti-fuzzing against pattern matching, we examine a number of code snippets that are repeatedly used throughout the protected binaries. We found that the injected code by AntiHybrid crafts several observable patterns, like hash algorithms or data-flow reconstruction code, and thus could be detected by attackers. One possible solution to this problem is to use existing diversity techniques to eliminate the common patterns [35]. We confirm that no specific patterns can be found in SpeedBump and BranchTrap because we leverage CSmith [66] to randomly generate a new code snippet for each FUZZIFICATION process.

Second, control-flow analysis can identify unused code in a given binary automatically and thus automatically remove it (*i.e.*, dead code elimination). However, this technique cannot remove our FUZZIFICATION techniques, as all injected code is cross-referenced with the original code. Third, data-flow analysis is able to identify the data dependency. We run protected binaries inside the debugging tool, GDB, to inspect data dependencies between the injected code and the original code. We confirm that data dependencies always exist via global variables, arguments, and the return values of injected functions. Finally, we consider an adversary who is capable of conducting manual analysis for identifying the anti-fuzzing code with the knowledge of our techniques. It is worth noting that we do not consider strong adversaries who are capable of analyzing the application logic for vulnerability discovery. Since FUZZIFICATION injected codes are supplemental to the original functions, we conclude that the manual analysis can eventually identify and nullify our techniques by evaluating the actual functionality of the code. However, since the injected code is functionally similar to normal arithmetic operations and has control- and data-dependencies on the original code, we believe that the manual analysis is time-consuming and error-prone, and thus we can deter the time for revealing real bugs.

7 Discussion and Future Work

In this section, we discuss the limitations of FUZZIFICATION and suggest provisional countermeasures against them.

Complementing attack mitigation system. The goal of anti-fuzzing is not to completely hide all vulnerabilities from adversaries. Instead, it introduces an expensive cost on the attackers' side when they try to fuzz the program to find bugs, and thus developers are able to detect bugs first and fix them in a timely manner. Therefore, we believe our anti-fuzzing technique is an important complement to the current attack mitigation ecosystem. Existing mitigation efforts either aim to avoid program bugs (*e.g.*, through type-safe language [32, 44]) or aim to prevent successful exploits, assuming attackers will find bugs anyway (*e.g.*, through control-flow integrity [1, 16, 30]). As none of these defenses can achieve 100% protection, our FUZZIFICATION techniques provide another level of defense that further enhances program security. However, we emphasize that FUZZIFICATION alone cannot provide the best security. Instead, we should keep working on all aspects of system security toward a completely secure computer system, including but not limited to secure development process, effective bug finding, and efficient runtime defense.

Best-effort protection against adversarial analysis. Although we examined existing generic analyses and believe they cannot completely disarm our FUZZIFICATION techniques, the defensive methods only provide a best-effort protection. First, if attackers have almost unlimited resources, such as when they launch APT (advanced persistent threat) attacks, no defense mechanism can survive the powerful adversarial analysis. For example, with extremely powerful binary-level control-flow analysis and data-flow analysis, attackers may finally identify the injected branches by BranchTrap and thus reverse it for an unprotected binary. However, it is hard to measure the amount of required resources to achieve this goal, and meanwhile, developers can choose more complicated branch logic to mitigate reversing. Second, we only examined currently existing techniques and cannot cover all possible analyses. It is possible that attackers who know the details of our FUZZIFICATION techniques propose a specific method to effectively bypass the protection, such as by utilizing our implementation bugs. But in this case, the anti-fuzzing technique will also get updated quickly to block the specific attack once we know the reversing technique. Therefore, we believe the anti-fuzzing technique will get improved continuously along the back-and-forth attack and defense progress.

Trade-off performance for security. FUZZIFICATION improves software security at the cost of a slight overhead, including code size increase and execution slow down. A similar trade-off has been shown in many defense mechanisms and affects the deployment of defense mechanisms. For example, address space layout randomization (ASLR) has been widely adopted by modern operating systems due to small overhead, while memory safety solutions still have a long way to go to become practical. Fortunately, the protection by FUZZIFICATION is quite flexible, where we provide various configuration options for developers to decide the optimal trade-off between security and performance, and our tool will automatically determine the maximum protection under the overhead budget.

Delay primitive on different H/W environments. We adopt CSmith-generated code as our delay primitives using measured delay on one machine (*i.e.*, developer's machine). This configuration implies that those injected delays might not be able to bring the expected slow down to the fuzzed execution with more powerful hardware support. On the other hand, the delay primitives can cause higher overhead than expected for regular users with less powerful devices. To handle this, we plan to develop an additional variation that can dynamically adjust the delay primitives at runtime. Specifically, we measure the CPU performance by monitoring a few instructions and automatically adjusting a loop counter in the delay primitives to realize the accurate delay in different hardware environments. However, the code may expose static pattern such as time measurement system call or a special instruction like `rdtsc`; thus we note that this variation has inevitable trade-off between adaptability and robustness.

8 Related Work

Fuzzing. Since the first proposal by Barton Miller in 1990 [40], fuzzing has evolved into a standard method for automatic program testing and bug finding. Various fuzzing techniques and tools have been proposed [57, 52, 29, 21, 34], developed [72, 37, 25, 23, 18, 9], and used to find a large number of program bugs [51, 72, 59, 26, 10]. There are continuous efforts to help improve fuzzing efficiency by developing a more effective feedback loop [6], proposing new OS primitives [64], and utilizing clusters for large-scale fuzzing [22, 24, 39].

Recently, researchers have been using fuzzing as a general way to explore program paths with specialties, such as maximizing CPU usage [49], reaching a particular code location [5], and verifying the deep learning result empirically [47]. All these works result in a significant improvement to software security and reliability. In this paper, we focus on the opposite side of the double-edged sword, where attackers abuse fuzzing techniques to find zero-day vulnerabilities and thus launch a sophisticated cyber attack. We build effective methods to hinder attackers on bug finding using FUZZIFICATION, which can provide developers and trusted researchers time to defeat the adversarial fuzzing effort.

Anti-fuzzing techniques. A few studies briefly discuss the concept of anti-fuzzing [63, 27, 41, 31]. Among them, Göransson *et al.* evaluated two straightforward techniques, *i.e.*, crash masking to prevent fuzzers finding crashes and fuzzer detection to hide functionality when being fuzzed [27]. However, attackers can easily detect these methods and bypass them for effective fuzzing. Our system provides a fine-grained controllable method to slow the fuzzed execution and introduces effective ways to manipulate the feedback loop to fool fuzzers. We also consider defensive mechanisms to prevent attackers from removing our anti-fuzzing techniques.

Hu *et al.* proposed to hinder attacks by injecting provably (but not obviously) non-exploitable bugs to the program, called "Chaff Bugs" [31]. These bugs will confuse bug analysis tools and waste attackers' effort on exploit generation. Both chaff bugs and FUZZIFICATION techniques work on close-source programs. Differently, our techniques hinder bug finding in the first place, eliminating the chance for an attacker to analyze bugs or construct exploits. Further, both techniques may affect normal-but-rare usage of the program. However, our methods, at most, introduce slow down to the execution, while improper chaff bugs lead to crashes, thus harming the usability.

Anti-analysis techniques. Anti-symbolic-execution and anti-taint-analysis are well-known topics. Sharif *et al.* [56] designed a conditional code obfuscation that encrypts a conditional branch with cryptographic operations. Wang *et al.* [62] proposed a method to harden the binary from symbolic execution by using linear operations instead of cryptographic functions. However, neither of them considered performance overhead as an evaluation metric. SymPro [7] presented symbolic profiling, a method to identify and diagnose bottlenecks of the application under symbolic execution. Cavallaro *et al.* [8] showed a comprehensive collection of evading techniques on dynamic-taint-analysis.

Software obfuscation and diversity. Software obfuscation transforms the program code into obscure formats that are difficult to analyze so as to prevent unexpected reverse engineering [12, 13]. Various tools have been developed to obfuscate binaries [15, 60, 33, 46]. However, obfuscation is not effective to impede unexpected fuzzing because it focuses on evading static analysis and the original program logic is still revealed at runtime. Software diversity instead provides different implementations of the same program for different execution environments, aiming to either limit attacks on a specific version (usually a small set of all distributions), or significantly increase the effort to build generic exploits [35, 3, 53, 50]. Fuzzing one of many diversified versions could be less effective if the identified bug is specific to one version (which is likely caused by an implementation error of the diversity mechanism). However, for bugs stemming from a programming mistake, diversity cannot help hinder attackers finding them.

9 Conclusion

We propose a new attack mitigation system, called FUZZIFICATION, for developers to prevent adversarial fuzzing. We develop three principled ways to hinder fuzzing: injecting delays to slow fuzzed executions; inserting fabricated branches to confuse coverage feedback; transforming data-flows to prevent taint analysis and utilizing complicated constraints to cripple symbolic execution. We design robust anti-fuzzing primitives to hinder attackers from bypassing FUZZIFICATION. Our evaluation shows that FUZZIFICATION can reduce paths exploration by 70.3% and reduce bug discovery by 93.0% for real-world binaries, and reduce bug discovery by 67.5% for LAVA-M dataset.

10 Acknowledgment

We thank the anonymous reviewers, and our shepherd, Stephen McCamant, for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA TC (No. DARPA FA8650-15-C-7556), and ETRI IITP/KEIT[B0101-17-0644], and gifts from Facebook, Mozilla and Intel.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[3] Algirdas Avizienis and Liming Chen. On the Implementation of N-version Programming for Software Fault Tolerance during Execution. *Proceedings of the IEEE COMPSAC*, pages 149–155, 1977.

[4] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, April 2005.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[7] James Bornholt and Emina Torlak. Finding Code that Explodes under Symbolic Evaluation. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018.

[8] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. Anti-taint-analysis: Practical Evasion Techniques against Information Flow based Malware Defense. Technical report, Stony Brook University, 2007.

[9] CENSUS. Choronzon - An Evolutionary Knowledge-based Fuzzer, 2015. ZeroNights Conference.

[10] Oliver Chang, Abhishek Arya, and Josh Armour. OSS-Fuzz: Five Months Later, and Rewarding Projects, 2018. https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html.

[11] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2018.

[12] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Science, University of Auckland, New Zealand, 1997.

[13] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[14] Timothy Garnett Derek Bruening, Vladimir Kiriansky. Dynamic Instrumentation Tool Platform. http://www.dynamorio.org/, 2009.

[15] Theo Detristan, Tyll Ulenspiegel, Mynheer Superbus Von Underduk, and Yann Malcom. Polymorphic Shellcode Engine using Spectrum Analysis, 2003. http://phrack.org/issues/61/9.html.

[16] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[17] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-scale Automated Vulnerability Addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[18] Michael Eddington. Peach Fuzzing Platform. *Peach Fuzzer*, page 34, 2011.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2018.

[20] GNU Project. GNU Binutils Collection. https://www.gnu.org/software/binutils, 1996.

[21] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2008.

[22] Google. Fuzzing for Security, 2012. https://blog.chromium.org/2012/04/fuzzing-for-security.html.

[23] Google. Honggfuzz, 2016. https://google.github.io/honggfuzz/.

[24] Google. OSS-Fuzz - Continuous Fuzzing for Open Source Software, 2016. https://github.com/google/oss-fuzz.

[25] Google. Syzkaller - Linux Syscall Fuzzer, 2016. https://github.com/google/syzkaller.

[26] Google. Honggfuzz Found Bugs, 2018. https://github.com/google/honggfuzz#trophies.

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

[28] Munawar Hafiz and Ming Fang. Game of Detections: How Are Security Vulnerabilities Discovered in the Wild? *Empirical Software Engineering*, 21(5):1920–1959, October 2016.

[29] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, August 2012.

[30] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, October 2018.

[31] Zhenghao Hu, Yu Hu, and Brendan Dolan-Gavitt. Chaff Bugs: Deterring Attackers by Making Software Buggier. *CoRR*, abs/1808.00659, 2018.

[32] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[33] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – Software Protection for the Masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 2015.

[34] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[35] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[36] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[37] LLVM. LibFuzzer - A Library for Coverage-guided Fuzz Testing, 2017. http://llvm.org/docs/LibFuzzer.html.

[38] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[39] Microsoft. Microsoft Previews Project Springfield, a Cloud-based Bug Detector, 2016. https://blogs.microsoft.com/next/2016/09/26/microsoft-previews-project-springfield-cloud-based-bug-detector.

[40] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[41] Charlie Miller. Anti-Fuzzing. https://www.scribd.com/document/316851783/anti-fuzzing-pdf, 2010.

[42] WinAFL Crashes with Testing Code. https://github.com/ivanfratric/winafl/issues/62, 2017.

[43] Unexplained Crashes in WinAFL. https://github.com/DynamoRIO/dynamorio/issues/2904, 2018.

[44] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[45] CSO online. Seven of the Biggest Recent Hacks on Crypto Exchanges, 2018. https://www.ccn.com/japans-16-licensed-cryptocurrency-exchanges-launch-self-regulatory-body/.

[46] Oreans Technologies. Themida, 2017. https://www.oreans.com/themida.php.

[47] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[48] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2018.

[49] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[50] Brian Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, (2):220–232, 1975.

[51] Michael Rash. A Collection of Vulnerabilities Discovered by the AFL Fuzzer, 2017. https://github.com/mrash/afl-cve.

[52] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[53] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(5):477–495, October 2012.

[54] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[55] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October–November 2007.

[56] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2008.

[57] 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 *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[58] Synopsys. Where the Zero-days are, 2017. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/state-of-fuzzing-2017.pdf.

[59] Syzkaller. Syzkaller Found Bugs - Linux Kernel, 2018. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md.

[60] UPX Team. The Ultimate Packer for eXecutables, 2017. https://upx.github.io.

[61] Daniel Votipka, Rock Stevens, Elissa M. Redmiles, Jeremy Hu, and Michelle L. Mazurek. Hackers vs. Testers A Comparison of Software Vulnerability Discovery Processes. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2018.

[62] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear Obfuscation to Combat Symbolic Execution. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS)*, Leuven, Belgium, September 2011.

[63] Ollie Whitehouse. Introduction to Anti-Fuzzing: A Defence in Depth Aid. https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2014/january/introduction-to-anti-fuzzing-a-defence-in-depth-aid/, 2014.

[64] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[65] Zhou Xu. PTfuzzer, 2018. https://github.com/hunter-ht-2018/ptfuzzer.

[66] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[67] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[68] Michal Zalewski. Fuzzing Random Programs without execve(), 2014. https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html.

[69] Michal Zalewski. New in AFL: Persistent Mode, 2015. https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html.

[70] Michal Zalewski. High-performance Binary-only Instrumentation for AFL-fuzz, 2016. https://github.com/mirrorer/afl/tree/master/qemu_mode.

[71] Michal Zalewski. Technical Whitepaper for AFL-fuzz, 2017. https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt.

[72] Michal Zalewski. American Fuzzy Lop (2.52b), 2018. http://lcamtuf.coredump.cx/afl/.

Appendix

A HonggFuzz Intel-PT-mode Result

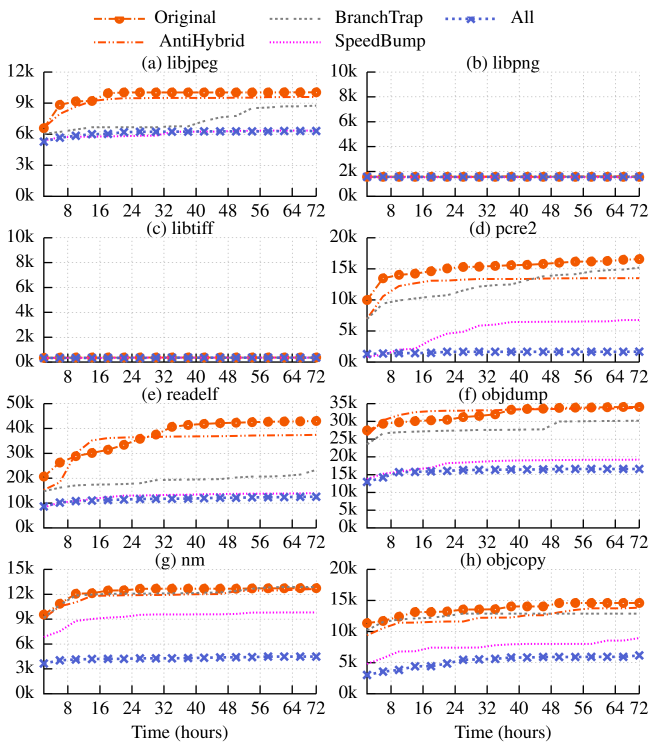

Figure 14: Paths discovered by HonggFuzz Intel-PT mode from real-world programs. Each program is compiled with five settings: original (no protection), SpeedBump, BranchTrap, AntiHybrid and all protections. We fuzz them for three days.

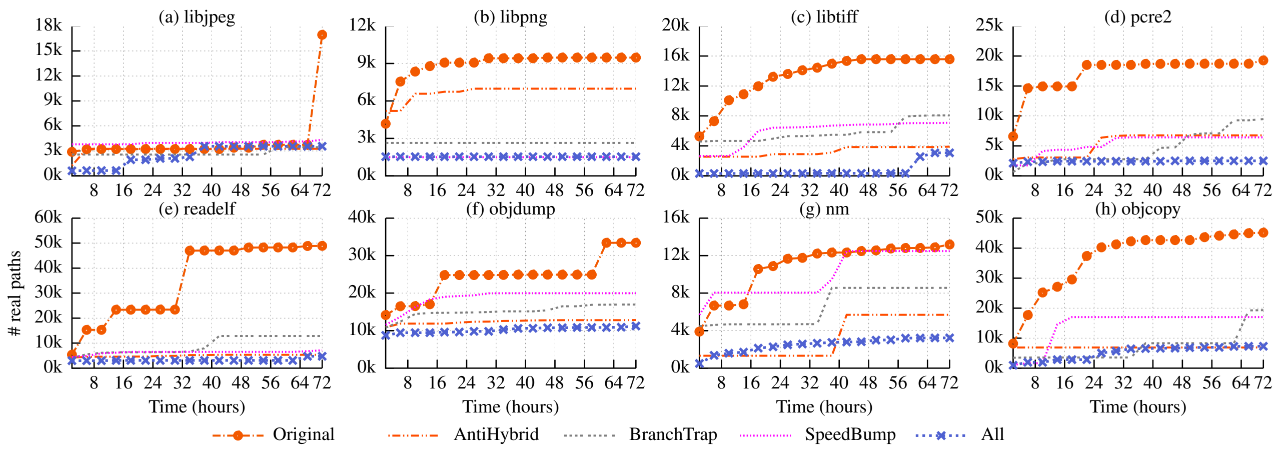

Figure 10: Paths discovered by QSym from real-world programs. Each program is compiled with the same five settings as in Figure 9. We fuzz these programs for three days, using QSym as the symbolic execution engine and AFL-QEMU as the native fuzzer.

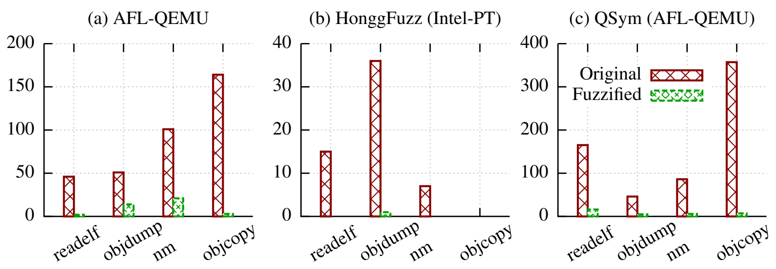

Figure 11: Crashes found by different fuzzers from binutils programs. Each program is compiled as original (no protection) and fuzzified (three techniques) and is fuzzed for three days.

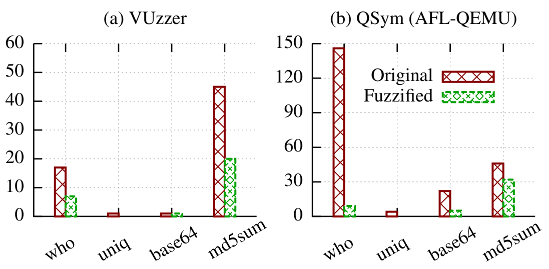

Figure 12: Bugs found by VUzzer and QSym from LAVA-M dataset. HonggFuzz discovers three bugs from the original `uniq`. AFL does not find any bug.