No-Fuzz: Efficient Anti-Fuzzing Techniques

Zhengxiang Zhou, Cong Wang, Qingchuan Zhao

City University of Hong Kong
{zxzhou4-c@my., congwang@, cs.qczhao@}cityu.edu.hk

Abstract. Fuzzing is an automated software testing technique that has achieved great success in recent years. While this technique allows developers to uncover vulnerabilities avoiding consequent issues (e.g., financial loss), it can also be leveraged by attackers to find zero-day vulnerabilities. To mitigate, anti-fuzzing techniques were proposed to impede the fuzzing process by slowing down its rate,
misinforming the feedback, and complicating the data flow.
Unfortunately, the state-of-the-art of anti-fuzzing entirely focuses on enhancing its defensive capability but underestimates the nontrivial performance overhead and overlooks the requirement of extra manual efforts.
In this paper, to advance the state-of-the-art, we propose an efficient and automatic anti-fuzzing technique and implement a prototype, called No-Fuzz.
Comparing to prior works, our evaluations illustrate that No-Fuzz introduces less performance overhead, i.e., less than 15% of the storage cost for one fake block.
In addition, in respect of the binary-only fuzzing, No-Fuzz can precisely determine the corresponding running environments and eliminate unnecessary storage overheads with high effectiveness.
Specifically, it reduces 95% of the total storage cost compared with the prior works for the same number of branch reductions.
Moreover, our study sheds light on approaches to improve the practicality of anti-fuzzing techniques.

Keywords: Anti-fuzzing · Software testing · Fuzzing.

1 Introduction

Fuzzing was first introduced as a software testing technique in 1990[25]. Typically, a fuzzer would persistently feed the target program with randomly generated inputs and observe the abnormalities of the program (e.g., segmentation faults) to identify program bugs. Recently, fuzzers have been well-developed - researchers integrate fuzzers with techniques like program instrumentation [36,28,31,29,3,12,38], program analysis techniques [33,30,35] for the efficiency in bug-finding. Besides, researchers also modify the fuzzing mechanisms of some classic works [36,3], to meticulously reallocate the fuzzing resources on some specific tasks (e.g., directly fuzzing a particular code area [7,8,10,23]). In general, fuzzers have achieved great success with plenty of bugs uncovered[14,27,15,31].

However, exposing bugs in the program is a double-edged sword. Developers can find and fix bugs before they spread on the internet. Meanwhile, attackers can also leverage fuzzers to explore zero-day vulnerabilities, which might cause financial loss to the companies. Although the adversaries can manually analyze the commercial software, recent studies [32,17] have shown that attackers lean more towards automated tools, like fuzzers, to find vulnerabilities than manual analysis. In the face of the worse situations of bug finding, anti-fuzzing techniques were proposed to hinder malicious use of fuzzers (ANTIFUZZ [16] and FUZZIFICATION [21]). The purpose of anti-fuzzing is to maintain the advantageous position of developers over adversaries on bug-finding. These techniques introduce penalties on fuzzers to disturb the fuzzing heuristics or slow down the fuzzing rate. The source codes of the protected program will be compiled into two versions - one is protected with anti-fuzzing codes, and the other is unmodified. Developers keep the original version, and they can thoroughly test the program. Adversaries only retrieve the protected version, and the anti-fuzzing codes inside the program can severely hinder the use of fuzzers. Consequently, developers are expected to uncover far more bugs than the adversaries and fix them to avoid the possible loss from zero-day vulnerabilities.

Anti-fuzzing techniques are promising, but the prototypes in prior works should be improved to more fine-grained application scope. Intuitively, the storage overhead should be taken into consideration for the practical adoption of anti-fuzzing techniques. In prior works, fake blocks are artificially inserted into the program to saturate the bitmap of fuzzers, while this technique can enlarge the size of the program by even several times the original program. Developers would be unwilling to burden such storage costs only for anti-fuzzing. Instead, they can resort to lighter obfuscation tools whenever applicable, even though these tools may not provide sufficient protection against fuzzers. Another factor that matters is the automation of the tools, i.e., existing prototypes are inconvenient to use. On the one hand, the developers have to locate some code areas manually; on the other hand, the prototypes have some dependencies with third-party tools/libraries that may be incompatible with the OS of users. We believe these factors challenge the future design and implementation of anti-fuzzing techniques. More specifically, the anti-fuzzing techniques should ideally hold the following two properties:

P1) *Both storage and performance overheads should be minimized;*

P2) *The implementation should support automation which has no modification to the development procedures of the program.*

Based on these thoughts, we propose our solution to anti-fuzzing techniques. The solution involves two categories of techniques - the passive detection methods and active disturbance methods. The passive detection methods precisely check whether the protected program is being fuzzed and launch mitigation strategies once fuzzers are detected. In our design, we integrate instrumentation checking and execution frequency checking to achieve low overhead anti-fuzzing techniques. The active disturbance methods impede fuzzers by attacking the basic fuzzing assumptions and prevent the fuzzers from working normally. We

optimize the defective fake blocks. In our design, the fake blocks achieve the minimum storage overhead, which is less than 15% of that in prior works.

We implement these techniques as a fully automated tool, i.e., No-Fuzz. The anti-fuzzing techniques are directly added to the source codes of programs, i.e., no modification is needed to the compilation procedures (e.g., header files, linked libraries, compilation commands). Notably, No-Fuzz is also compatible with other anti-fuzzing techniques in prior works that are not mentioned.

In the evaluation, we measure the effectiveness of our techniques in reducing branch coverage with real-world software from Binutils and two popular benchmarks (Google FTS[1] and Magma[18]). Moreover, we show that our techniques can hinder bug findings for the LAVA-M [13] dataset. To confirm our optimizations to prior works, we have also compared No-Fuzz with the corresponding techniques in ANTIFUZZ[16] and FUZZIFICATION[21]. The results show that our design introduces less overhead and mitigates the negative effects of anti-fuzzing techniques on regular users. Specifically, we reduce about 95% of the storage cost compared with the prior works for the same number of branch reductions. Furthermore, we tackle the obstacle that there is no suitable metric to compare different anti-fuzzing techniques currently. Existing works measure the anti-fuzzing effects and the overhead separately. However, the performance and overhead are orthogonal - both of them vary according to different configurations; it is unfair to compare the performance of different works with unequal overhead directly. Therefore, in addition to measuring the performance and overhead separately, we propose a new metric - "anti-fuzzing efficacy" linking the two metrics to measure the increased defensive capability per unit overhead.

In short, this paper makes the following contributions: 1) throws light on the facts of existing anti-fuzzing prototypes and summarises the properties of the ideal anti-fuzzing techniques; 2) designs and implements automated anti-fuzzing prototype No-Fuzz which can detect and disturb run-time fuzzing mechanisms; 3) evaluate No-Fuzz and some of the prior anti-fuzzing techniques on common benchmarks, showing No-Fuzz's negligible overhead to the protected binary and its effectiveness at impeding binary-only fuzzing. The source codes of all implemented tools are available at https://github.com/CongGroup/No-Fuzz.

2 Technical Background of Anti-fuzzing

The purpose of anti-fuzzing techniques is to combat fuzzers to reduce the number of bugs reported on protected binaries. Existing techniques can be majorly summarized as anti-fast-execution, anti-feedback, and anti-hybrid techniques based on the affected fuzzing mechanisms. We will briefly introduce them in the following parts of this section.

Anti-fast-execution: introduce latency to binary. One of the fuzzers' assumptions is that more trials with different inputs are expected to explore more paths in the binary. Fuzzers are usually designed with accelerating techniques feeding thousands of seeds per second to the program under test (PUT) [37]. Anti-fast-execution techniques insert latency into the binary to prevent

fast-execution. However, the challenge is that the latency can also affect regular users. ANTIFUZZ[16] inserts delay functions in the error handling codes manually; FUZZIFICATION [21] inserts the latency functions in cold blocks. Both techniques are trying to delay the areas that regular users rarely reach, but fuzzers are easy to fall into.

Anti-feedback: disturb the feedback information. Modern fuzzers majorly rely on two feedbacks to decide fuzzing heuristics - coverage-feedback and error signals. The coverage information is stored in a bitmap of limited size, and fuzzers are expected to make decisions on seeds and mutations to maximize the coverage. The error signals inform fuzzers to save and report the seeds triggering bugs, which is also the ultimate goal of using fuzzers.

Anti-feedback techniques insert fake blocks into the protected binary to disturb the coverage feedback. These blocks contain codes irrelevant to the program logic but are recorded as valid blocks in the bitmap. If most space of the bitmap has been taken up by these fake blocks, fuzzers will be unable to update new coverage. ANTIFUZZ hijacks the control flows to randomly generated fake functions in the protected program. FUZZIFICATION inserts a fixed number of constraints and functions into the binary, and builds ROP chains as fake paths on the assembly code snippets.

For the error signals, ChaffBugs[19] suggests inserting non-exploitable bugs into the binary to confuse the segmentation faults reported to fuzzers. ANTIFUZZ proposed an approach to hinder the crash discovery by installing a signal handler. The handler hides signals from fuzzers with elegant exits, and thus fuzzers are unaware of crashes.

Anti-hybrid: impede program analysis. Hybrid fuzzers[30,35,20,11] generally rely on taint analysis and symbolic execution to accelerate fuzzing. Anti-hybrid techniques embed complex data flows in protected binary to hinder both of the techniques. The idea is based on the fact that program analysis techniques have difficulty in dealing with complex data flows due to the limited CPU resources. ANTIFUZZ encrypts and decrypts the inputs and transforms variables in critical comparisons to their hash values. Similarly, FUZZIFICATION adds extra copy operations to the operand string to complicate the data flows and mislead taint analysis engines to a wrong tag map.

3 No-Fuzz Design

No-Fuzz includes passive detection methods and the optimized active technique (fake blocks) of prior works. Passive detection methods detect whether the protected binary is being fuzzed in binary-only-fuzzing (BOF) mode. Once the fuzzers are found, the protection will trigger fuzzing mitigation mechanisms (e.g., introducing latency). As aforementioned, the active methods in prior works are not practical due to the storage overhead. We optimize the fake blocks and design the landing space exploiting the block-identification mechanism of binary-only instrumentation. It reduces the storage overhead of a fake block to only one byte.

3.1 Passive detection methods

A fundamental requirement for anti-fuzzing is to avoid the negative effects of the inserted anti-fuzzing techniques on regular users. Ideally, if the protected program itself can accurately perceive the fuzzers, we can impose severe penalties on adversaries covertly. Based on this thinking, we introduce passive detection methods to identify the running environments of protected binaries. Once fuzzers are detected, we carry out mitigation mechanisms such as delaying the execution and aborting the program to prevent fuzzing.

Detect binary-only instrumentation In the scenario of using anti-fuzzing techniques, adversaries are not able to retrieve the source codes of the protected binary - they rely on the binary-only mode of fuzzers. The key is that no matter what techniques they are using, they have to collect coverage information of the target program. Techniques such as dynamic instrumentation, hardware assistance, and binary rewriting are the most used for this observation, but all of them cause significant latency (a timing gap) to the PUT. We can detect the timing gap between the native execution and the execution with coverage-collecting codes to determine the running environment.

Timing-related techniques are common in the scope of malware detection [22,24,5]. We learn from existing works and design the detection on binary instrumentations. In the native execution environment (real CPU), the control flow directly falls into the block after a branch-taken instruction. On the contrary, with BOF, the instrumented program executes the additional instructions collecting coverage at the beginning of a block. We detect BOF by checking the edge instructions count (EIC). EIC is the estimated number of instructions executed when entering a function or a block (instructions of an edge). According to the experiments, the EIC for BOF can be about ten times larger than that in the native execution. If we detect the timing gap in a protected program, we acknowledge the existence of BOF and carry out mitigation techniques.

Mitigation - Introduce latency. Due to the performance variation of the CPU, there will be false positives in the detection. A few portions of executions can have a relatively large timing gap, even if they are in the native environment. This usually happens when the CPU is conducting context switching, and extra overhead is counted as a part of the timing gap. According to our experiments, 0.03% of the executions are false positives in a stable environment, and the false-positive rate will be 0.1% in a busy environment where too many parallel tasks are executed simultaneously.

As a consequence, the mitigation mechanisms to fuzzing should be moderate in case the false positives affect regular users. We add a one-second latency to the program by triggering an IO blocking if the BOF instrumentation is detected. Although one second is insufficiently long for a fuzzer, the general effectiveness of the penalty can be guaranteed if we insert more than one detection function into the protected program.

Examining execution frequency The nature of fuzzing is that the PUT will be re-executed a large number of times in a short period. This can be leveraged to detect whether the program is being fuzzed. We come up with the idea from "many a little makes a mickle". If the PUT leaves some vestiges every fuzzing round, the vestiges will accumulate during the fast re-executions. After a while, they grow large enough and inform the PUT that it is being fuzzed. In our design, the protected program creates a temporary file every time it runs. If more than 60 files are created in a minute (the threshold is according to configurations), the program will be alerted to the existence of a fuzzer. However, the challenge is that the program only creates the files, but it is difficult to manage them. It can be time-consuming to traverse the temporary files and check which are created by the protected program. Besides, not deleting them can mess up the file systems for regular users. To cope with this program, we find that the daemon process is suitable for the management of temporary files.

A daemon process is a process that runs as a background process. It detaches from the parent process and keeps running after the termination of its parent. We designed the daemon process to patrol the temporary files - to prevent the temporary files from being unintentionally deleted and delete them after the patrolling. The patrolling daemon process detaches itself from the protected program during the execution. A temporary file with an ID to indicate its order will be created by the daemon process. These files are created in ascending order, and the largest order is the detection threshold for fuzzers. It then locks the file for a period which we call "patrolling time". After the patroling, it checks whether the file it locks is correct and deletes the file it creates. The protected program seeks temporary files with the threshold order for every execution. Once the file is found, it means the program has been executed more than the threshold number of times in the patroling time, and the BOF is likely to exist; so we can apply the mitigation techniques.

Mitigation - Aborting program. Different from the timing gap, the results of the daemon process are accurate, and there are no false positives. We can adopt a more severe penalty in this method. The protected PUT can abort the execution or trigger an artificially inserted bug to misinform the crashes to fuzzers. To further avoid the mitigation strategy affecting the regular users in some unexpected situations, the developers can set a long patrolling time (5 minutes) and a large threshold (1000 files). Regular users can rarely execute the program at such a high frequency.

3.2 Active methods: Minimum fake blocks

Existing fake blocks impede fuzzers at a non-optimal storage cost. AntiFuzz[16] (default configuration) introduces about 20 MB of anti-fuzzing codes, while based on the OSS-Fuzz[4] project, most commercial software occupies no more than 100 MB. Besides, small programs are more sensitive to storage costs and are hard to burden high storage costs. Unfortunately, they are more likely to be chosen as the fuzzing targets by attackers due to their faster execution speed and less program logic.

This defect comes to the fore as future fuzzers enhance the capability of fuzzing mechanisms (e.g., size of the bitmap) or improve fuzzing heuristics (e.g., scheduling seeds to avoid triggering anti-fuzzing mechanisms). Anti-fuzzing techniques have to insert more protection codes in proportion to the increased fuzzer capability to handle the intensified arms race. Under this condition, the fundamental storage overhead should be as low as possible; otherwise, the proportioned overhead needed in the arms race will be unsustainable.

The landing space is designed to minimize the extra storage overhead of fake blocks. Existing approaches pile up function calls and constraints, disturbing fuzzers at the function level, where one fake block takes up about nine bytes (one *cmp* and one *jmp* instructions). However, we observe that some of the storage overhead of fake blocks is unnecessary at the assembly level. For example, C compilers generate function frames for each function that controls the base pointer and stack pointer (e.g., *push ebp*). These assembly codes have nothing to do with anti-fuzzing, and eliminating these codes can further reduce the storage overhead.

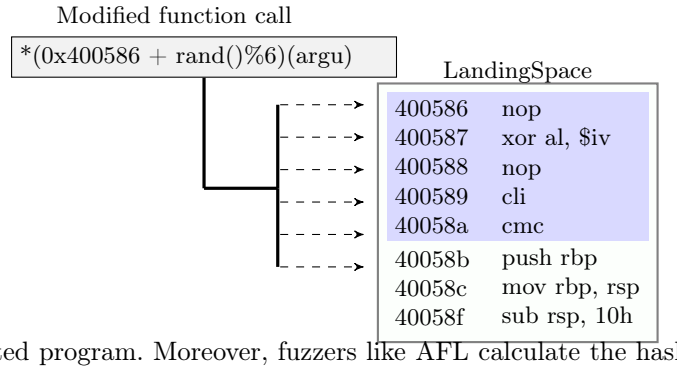
As adversarial cannot retrieve the source codes of targets, they use BOF with the assistance of external tools to collect coverage feedback. These tools insert codes before entering a new block. If a control-flow-changing instruction (*jmp*, *call* and *ret*) is encountered, they generate a new block record as the updated coverage. Theoretically, the minimum block is the instruction only occupying one byte (opcode of the minimum size), which should be at most 15% (from nine bytes to one byte) of the storage cost of prior works. To achieve the minimum fake block, we instrument each function with a code segment called *landing space*. The landing space contains instructions that have no effect on the normal execution. They are either one-byte instructions or two-byte instructions with an immediate value which is the opcode of a one-byte instruction. This ensures that each byte in the landing space can be translated into a valid instruction.

We further modify the destination address of function calls to the address of a random byte in the landing space. The rationale is that when the modified function call is invoked, the control flow "land" at a random instruction. The fuzzer considers this instruction the start of a new block and records the address as new coverage. Since the control flow can "land" at any byte in the landing space, fuzzers will record most of the possible addresses after sufficient rounds of fuzzing. The corresponding fake coverage can overwhelm the fuzzer's bitmap.

Figure 1 illustrates the assembly codes of a function and the landing space. The original destination address of the function is *0x40058b*, and we insert the landing space before this address in the text section between *0x400586* and *0x40058a*. The modified function call (*0x400586 + rand()%6*) jumps to the landing space or the original start of the function. In this example, a fuzzer will record six fake blocks at the cost of six bytes.

Optimizations. The naive implementation of the landing space seems able to disturb the coverage feedback of BOF. However, we found that the size of the landing space is restricted. A too large size introduces non-negligible latency to

Fig. 1: A function with landing space.$iv is the immediate value, in this example, it will be 0x90 which is the opcode for nop.



the protected program. Moreover, fuzzers like AFL calculate the hash value by exclusive-or operations on the addresses of two blocks. The problem is that the addresses of fake blocks in the landing space are close, and so are the calculated hash values. The chance of hash collision in the landing space is higher than that in normal functions. Intuitively, the more hash collisions happen in the landing space, the less bitmap can be saturated by fake blocks. In other words, the fuzzer will be more powerful in discovering real branches in the protected program. As a means of coping strategy, we propose two optimizations to mitigate the limitations.

Jump over unnecessary instructions. If the control flow lands on the first few bytes in the landing space, it has to execute the rest instructions, which incurs significant latency for a large landing space. To avoid executing the unnecessary instructions, we modify some one-byte instructions to a short jump, and the jumping offset is the opcode of the next instruction. As shown in Figure 2, if control flow lands at *0x400500*, the assembly code is translated as a two-byte short jump with offset *0x36*. However, if it lands at the next byte *0x400501*, the corresponding assembly code is "xor al, 0x90". The functionality of the landing space still remains as every byte can be disassembled correctly and recorded as a new block. Yet the jump instructions reduce the performance overhead to 5% of the original landing space.

Spray LandingSpace at different addresses. To reduce the hash collision rates, we increase the blocks of different addresses. We wrap functions in the original program with several intermediate blocks. The intermediate blocks only redirect the control flows in the protected binaries but have no effect on the program execution. We artificially keep wide disparities in the addresses of intermediate blocks; thus, these blocks are likely to generate more hash values than those generated from a single landing space. The size of the landing spaces in these functions is accordingly reduced, and they will be distributed to the intermediate blocks.

4 Evaluation

We evaluate No-Fuzz to answer the following four research questions (**RQ**s):

– **RQ 1.** Can No-Fuzz hinder fuzzers from exploring new branches?

Fig. 2: Jump over unnecessary blocks

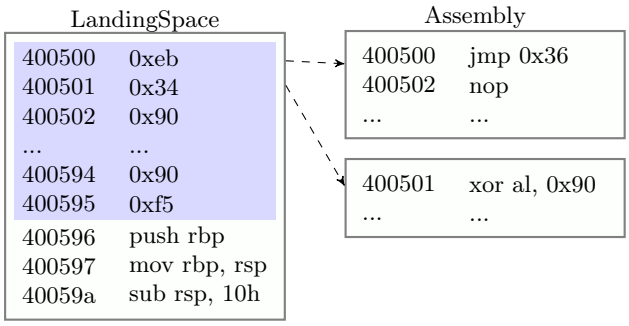
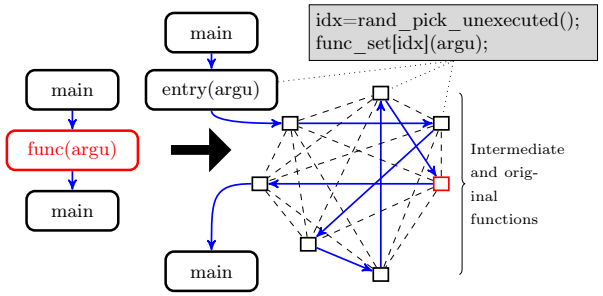


Fig. 3: Spray intermediate functions in protected binaries



– **RQ 2.** How effective are the anti-fuzzing techniques at preventing fuzzers from finding bugs?
– **RQ 3.** What are the storage and performance overhead to deploy anti-fuzzing techniques?
– **RQ 4.** What is the suitable metric to compare different anti-fuzzing techniques?

For **RQ 1**, coverage is considered orthogonal to the bug-finding abilities of fuzzers [9]. The more coverage a fuzzer can reach, the more likely it finds bugs inside the target program. We evaluate the coverage reduction on real-world binaries after applying No-Fuzz to show the defense effectiveness of anti-fuzzing techniques. To stress **RQ 2**, we evaluate the anti-fuzzing techniques on the LAVA-M[13] benchmark and measure the shortest time needed to find a bug. The benchmark consists of four buggy binaries (`base64, md5sum, who, uniq`) with dozens to thousands of artificially inserted bugs.

The **RQ 3.** and **RQ 4.** are both related to the overhead evaluation of anti-fuzzing techniques. To answer **RQ 3.**, we evaluate the storage and performance overhead of anti-fuzzing techniques on real-world programs of different sizes. The **RQ 4.** is based on the concern that the overhead alone is not able to judge an anti-fuzzing technique comprehensively. The problem is that if the number of defensive codes added to the protected programs increases, the overhead is likely to increase accordingly. The defensive capability is orthogonal to the extra

overhead as the more defensive codes are inserted into the protected binary, the safer it is able to be. Thus judging a defensive technique only based on one metric (anti-fuzzing effect or the overhead) is not enough. We suggest combining these two metrics and unifying the judging criteria of anti-fuzzing techniques by measuring the defensive ability at a unit cost of storage or execution rate. We define a new metric anti-fuzzing efficacy as the number of reduced branches per byte of extra storage cost and that per millisecond of latency. It measures the capability of anti-fuzzing techniques with respect to the introduced overhead.

In all experiments, the latency mitigation is set to be one second; the daemon process will patrol for one minute and alert if there are more than 60 times of executions; the landing space is configured to occupy 100 bytes, and the functions will be wrapped in 50 intermediate functions.AFL and AFL-based fuzzers (AFLFast and QSYM) use AFL-QEMU. HonggFuzz supports Intel-PT and QEMU, and we adopt both binary-only modes. Each fuzzing campaign runs with three CPU cores. Notably, QSYM runs two AFL instances with two CPU cores and an SMT solver using one core. Fuzzing campaigns on the LAVA-M dataset are kept running for 48 hours, while the others last for 24 hours. Due to the nondeterministic fuzzing behaviors, we repeat fuzzing campaigns ten times for each fuzzer x target combination.

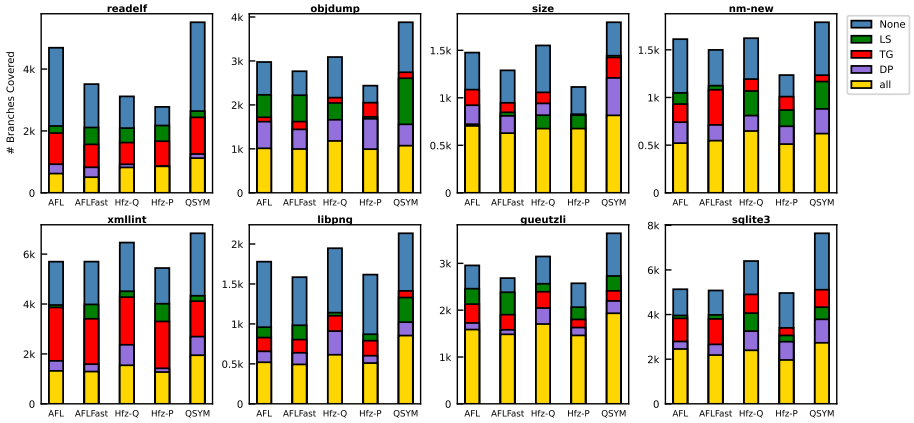
4.1 Reducing code coverage

We evaluated the branch coverage of five fuzzers against eight real-world binaries from Binutils, Magma, and Google FTS. Figure 4 shows the average branches covered by each fuzzer on the binaries with and without No-Fuzz protection. Each technique is separately evaluated to avoid the effect of one technique covering up others. From the figure, the combo of all No-Fuzz techniques can severely hinder the branch explorations of fuzzers. The fuzzers can only discover 36.9% of branches on average that should have been, and most of the branches are just for initializations and input correctness checks.

A single passive detection technique reduces 34.8% - 52.2% of the branch coverage. The effect variation of the detection should also be attributed to the choices of mitigation techniques and the difference in fuzzers. As the results show, aborting the PUT (the purple columnar) is more effective than introducing latency (the gray ones) to the protected programs. However, introducing latency affects users more slightly than aborting the program. This is the trade-off between effectiveness and impacts on users. It will be reckless to conclude that one mitigation technique outperforms the others. Our suggestion is to apply the more severe mitigation techniques to the more precise detection techniques.

The landing space hinders, on average, 32.4% of the branches. We observe that it is less effective against HonggFuzz. It is because Honggfuzz records the coverage in a temporary file and the size of the bitmap is 16M, while the bitmap of AFL only occupies 64K. We consider the current configuration of the landing space too small to saturate the bitmap of Honggfuzz. In the actual situation, the developers can configure a larger landing space for better protection against Honggfuzz.

Fig. 4: Branch covered by four fuzzers against eight binaries with and without different protections. **T**he techniques are **T**iming **G**ap, **D**aemon **P**rocess, **L**anding **S**pace. The fuzzers are AFL, AFLFast, **H**onggf**u**z**z**-**Q**EMU,**H**onggf**u**z**z**-**P**T, and QSYM .



4.2 Preventing fuzzers from finding bugs

LAVA-M benchmark. Despite many recent works suggesting using the up-to-date benchmarks in bug-finding experiments(e.g., Google fuzzer test suite[1] and Magma[18]), we find they are not suitable for the evaluation of BOF. The problem is that these benchmarks heavily rely on sanitizers, but unfortunately, most BOF techniques are not able to support sanitizers. There are some works like QEMU-AddressSanitizer[2] to fill the gap in BOF and sanitizers, but our evaluation covers different BOF techniques, and not all of them have such complementary tools. Due to this limitation, we decided to use LAVA-M, whose bugs directly trigger segmentation faults and can be caught by the BOF.

Moreover, the efficiency of BOF severely degrades compared with static instrumentation, which is almost a quarter of that of the latter. Even the LAVA-M benchmark contains thousands of bugs; only a few unique bugs can be uncovered for each buggy binary in BOF. Measuring the number of bugs found will be insignificant even though it is considered the ground truth for fuzzer evaluation. Instead of the number of bugs found, we measure the time that fuzzers need to find the first bug in each buggy binary within 48 hours. This metric can better illustrate the bug-finding capabilities of BOF in the LAVA-M benchmark.

Results. The average time of five fuzzers to find one bug in the LAVA-M benchmark is shown in Table 1. From the table, we find that all fuzzers are able to find at least one bug in the unprotected programs in 48 hours. Note that QSYM finds the bug in only several minutes, which is quite faster than other fuzzers. It can be attributed to the design of the LAVA-M benchmark. The bugs in LAVA-M are all designed based on an integer comparison. If an input can bypass the comparison, the corresponding bug will be triggered. This mechanism is essentially more beneficial to fuzzers that are able to solve constraints. It makes

sense that QSYM , as a hybrid fuzzer, outperforms other mutational fuzzers due to its symbolic execution engine.

On the other hand, with the anti-fuzzing defenses, some fuzzing campaigns exceed the 48-hour time limit and fail to find any bug. The rest campaigns uncover some bugs, but it takes much more time than that spent on the corresponding unprotected programs. Notably, we found Honggfuzz is incompatible with the target md5sum in LAVA-M as it misjudges the handled errors as crash signals. HonggFuzz generates millions of crash seed files, and the majority of them are false positives. It is difficult to distinguish the correct crash seeds in the millions of files, and we have to discard this fuzzer x target combination.

Generally, the passive detection techniques and the landing space successfully impede all fuzzing campaigns, as the time to find a bug extends after applying these techniques. If all of the anti-fuzzing techniques in No-Fuzz are applied, none of the fuzzing campaigns can find a bug. Overall, the evaluation confirms that No-Fuzz is effective at preventing the BOF of different fuzzers from discovering bugs in the protected programs.

Table 1: Time of fuzzers to find a bug in native and protected LAVA-M. √ means the fuzzing campaign fails to find a bug within 48 hours.

	native	TG	DP	LS	all
base64					
AFL	12h54m	√	√	√	√
AFLFast	13h8m	√	√	√	√
Hfz-QEMU	1h22m	√	√	√	√
Hfz-PT	5h23m	√	√	√	√
QSYM	2m	√	√	√	√
md5sum					
AFL	36h32m	√	√	√	√
AFLFast	9h20m	√	√	√	√
Hfz-QEMU	-	-	-	-	-
Hfz-PT	-	-	-	-	-
QSYM	51m	√	√	√	√
who					
AFL	3h8m	√	√	√	√
AFLFast	6h37m	√	√	√	√
Hfz-QEMU	37m	2h9m	√	6h32m	√
Hfz-PT	4h31m	11h45m	√	√	√
QSYM	1m	√	√	√	√
uniq					
AFL	7h19m	√	√	√	√
AFLFast	6h47m	23h59m	√	√	√
Hfz-QEMU	4m	10h55m	√	9h3m	√
Hfz-PT	2h48m	16h20m	√	17h56m	√
QSYM	5m	√	√	√	√

4.3 Performance & storage overhead of No-Fuzz

We are inspired by the fact that the size of input files can affect the performance overhead accordingly. Generally, a larger input will invoke more functions and be processed for a longer time. Considering the overhead of the landing space is also accordingly proportional to the functions executed, for fairness, we prepare two sets of input files. One set only contains small invalid files, which can fastly trigger errors in the programs, while the other set consists of valid samples of different sizes to trigger the normal functionalities. The results adopt the average

time of the executions with both sets of input samples. Another consideration is that the overhead will be less significant for large and complex programs. The programs are categorized into two groups based on their size and average execution time to mitigate the bias in the evaluations, as shown in Appendix A3. The evaluation results of each group will be analyzed separately.

Performance overhead. As shown in Table 2, the performance overhead for passive detection techniques is about 10-20% for small binaries and less than 1% for large binaries. Although the overhead is relatively large for small programs, the absolute latency is only around 5 ms, which is usually unnoticeable for regular users. The timing gap detection introduces a little more latency than the daemon process. We think it should be attributed to the false positives of the timing gap. Similarly, the landing space introduces the overhead proportioned to the size and complexity of the programs. Small programs incur 40% latency, while for large programs, the proportion decreases to 2%.

Storage overhead. From Table 2, No-Fuzz introduces negligible storage overhead to protected binaries. The passive detection techniques take up storage ranging from about 1KB to 50KB (10KB on average), but they are all less than 1% of the original size of the protected programs. The landing space inserts fake blocks according to the number of functions in the protected binary. Basically, the fewer functions in the original program, the less overhead it has. The storage cost can range from 0.3MB to 1MB (0.8M on average) for different programs.

Comparisons with prior works. We evaluate the existing anti-coverage techniques in prior works to show that the landing space is worth it. The default configurations of ANTIFUZZ and FUZZIFICATION insert a fixed number of fake blocks; thus the storage overhead is stable - 20MB for ANTIFUZZ and 1.2MB for FUZZIFICATION . Clearly, both techniques take up more space than the landing space, and the storage advantage of No-Fuzz is more evident for small binaries due to the accordingly fewer fake blocks in smaller binaries. Note that the storage overhead of FUZZIFICATION is much smaller than that of ANTIFUZZ . However, according to our experiments (Appendix A1), we find the default configuration of the FUZZIFICATION is not enough to saturate the bitmaps of fuzzers. The real storage overhead of the effective configuration of FUZZIFICATION should be even larger than the current 1.2 MB.

Table 2: Overhead(CPU) of No-Fuzz and anti-coverage techniques of **A**NTI**F**UZZ and **F**U**ZZ**IFICATION on real-world programs.

	TG	DP	LS	All	AF(cov)	FZ(cov)	Reference
CPU							
Small	6.8ms(21.4%)	3.4ms(10.6%)	13.1ms(40.9%)	47.2ms(147.5%)	11.3ms(35.3%)	14.7ms(45.9%)	32.0ms
Large	23.5ms(1.1%)	10.7ms(0.5%)	43.8ms(2.0%)	64.7ms(3.0%)	15.6ms(0.7%)	44.6ms(2.1%)	2156.2ms
Storage							
Small	8.4K(0.2%)	10.5K(0.3%)	0.8M(25.9%)	1.1M(35.3%)	22.2 M(696.6%)	1.25 M(39.1%)	3.2M
Large	43.0K(0.04%)	27.1K(0.02%)	2.0M(1.9%)	2.4M(2.2%)	22.3 M (21.0%)	1.27 M (1.2%)	106.5M

4.4 Anti-fuzzing efficacy

To compare the merits of different anti-fuzzing techniques, we introduce a new metric - anti-fuzzing efficacy. The efficacy illustrates the associations between

the anti-fuzzing effects (coverage reduction) and storage/performance overhead. As anti-fuzzing techniques improve the defensive capability by inserting extra codes into the protected programs, more defensive codes promise more defensive effects. A well-designed anti-fuzzing technique should introduce as low overhead as possible while possessing effective defensive capability. The defense capability per unit storage/performance cost should be a better metric to describe the anti-fuzzing effects. Specifically, we calculate the efficacy by the number of coverage reductions every kilobyte of defensive codes and the reduction of every millisecond of latency. As a reference, we also calculate the efficacy of ANTIFUZZ and FUZZIFICATION based on the results in Appendix A1.

From Table 3, the passive detection techniques have both the highest performance and storage anti-fuzzing efficacy. Passive detection techniques are the most affordable anti-fuzzing techniques that can hinder the BOF at the lowest cost. Moreover, these techniques insert fixed defensive codes into the protected programs. Thus, the efficacy is stable and more or less has the same order of magnitude among all evaluated programs. The performance efficacy of the landing space is about 20% of the passive detection techniques, and the storage efficacy is much smaller (only about 1% of the passive detection methods). Despite the fact that the landing space seems to be less efficient, it can be deemed as a complement to the passive detection techniques. As we will discuss in Section 5, a single defensive technique is weak against adversaries, and it is worth the development of different techniques.

For reference, we evaluate the anti-coverage techniques of ANTIFUZZ and FUZZIFICATION . The performance efficacy of landing space and ANTIFUZZ is close, while FUZZIFICATION is inefficient due to the insufficient number of blocks. The storage efficacy of the landing space is 14-28 times that of these anti-coverage techniques, i.e., on average, we reduce about 95% of the initial storage cost. It illustrates that No-Fuzz is more practical, which can better utilize the storage while keeping adequate anti-fuzzing protection.

Table 3: Space and performance efficacy of different anti-fuzzing techniques against four fuzzers.

	TG	DP	LS	All	AF(cov)	FZ(cov)
Performance (#branches / ms)						
AFL	183.9	559.3	84.2	46.6	144.1	11.5
AFLFast	165.0	509.3	61.9	42.3	121.7	5.3
Hfz-Q	158.5	530.1	86.2	47.0	133.9	17.0
Hfz-P	134.5	428.4	62.6	36.9	91.9	5.0
QSYM	226.8	684.9	120.7	58.6	174.6	14.8
Avg	173.7	542.4	83.1	46.3	133.2	10.7
Storage (#branches / kB)						
AFL	148.8	181.1	1.4	2.0	0.07	0.14
AFLFast	133.6	164.9	1.0	1.8	0.06	0.15
Hfz-Q	128.3	171.6	1.4	2.0	0.07	0.20
Hfz-P	108.9	138.7	1.0	1.6	0.05	0.06
QSYM	183.6	221.8	2.0	2.5	0.09	0.17
Avg	140.6	175.6	1.4	2.0	0.07	0.14

5 Discussion

While our design performs effectively and efficiently, it could be further improved. We consider our design a complement to prior works, and we appreciate some designs in prior works which also deserve adoption in the future development of anti-fuzzing tools (e.g., installing the signal handler to hide crashes). We will also stress these concerns in our discussion. In the following, we will discuss the robustness of the anti-fuzzing technique and the advantages of anti-fuzzing over obfuscation because the potential inherent problems (i.e., robustness) and the seemingly possibility that they can be substituted by obfuscation techniques in certain cases.

Robustness of anti-fuzzing techniques. A primary concern about anti-fuzzing is its robustness. In particular, a dedicated attacker can perform manual analysis to disarm the defense if the details are known, and several prior works suggested applying obfuscation techniques as a countermeasure against reverse engineering. However, for experienced attackers, obfuscation techniques can also be disarmed. We want to reflect the necessity for anti-fuzzing techniques to be robust against manual analysis. Anti-fuzzing techniques are proposed to introduce extra efforts (time, resources, knowledge, etc.) adversaries need to fuzz a protected program. Particularly, they are suitable to defend the untargeted fuzzing tasks on a large scale that are not worth enough to analyze a single binary for attackers manually. Moreover, anti-fuzzing techniques enhance the basic requirements of BOF, from knowing nothing to at least understanding anti-fuzzing and reverse-engineering techniques. The defensive techniques can further reduce the chance that protected binaries are chosen as the targets of BOF. Due to the above reasons, we consider that reverse engineering does not contradict the ultimate purpose of anti-fuzzing techniques.

Anti-fuzzing or obfuscation. Obfuscation is originally considered a potential solution to anti-fuzzing. Compared with the emerging anti-fuzzing techniques, it is well-developed with profound community support. Prior works have conducted some experiments to show the ineffectiveness of obfuscation in anti-fuzzing[21,16]. However, we will revise their arguments with more experiments and show that obfuscation techniques can be effective at times.

In fact, there are already obfuscation techniques designed to confront symbolic executions, which are similar to anti-hybrid techniques [6,34]. In addition, the obfuscation techniques involving self-modifying codes can be even more powerful against BOF. The self-modifying code is a common technique used in packing and encryption. It reuses the memory space by overwriting the existing opcodes with those of new instructions. The problem is that no matter how many functions are overwritten to a self-modifying block, they possess the same memory address. Most fuzzers record the hash values of function block addresses, and the self-modifying block will be identified as only one function; thus, the coverage information of overwritten functions is lost.

We have conducted some experiments with BOF and dummy programs where the programs are protected by self-modifying codes. The results (appendix A4 and A4) show that BOF cannot be correctly performed on the program with self-

modifying codes. Besides, [26] shows that self-modifying codes will severely slow down the translations in emulators, which confirms the anti-fuzzing effectiveness of obfuscation against BOF like `afl-qemu` and `honggfuzz-qemu`. Fortunately, self-modifying codes are usually not welcomed by developers. Commercial software rarely uses self-modifying codes due to the possibility of false positives as malicious attempts and the new bugs introduced by risky modifications. Overall, anti-fuzzing techniques cover the shortage of static obfuscation techniques against fuzzers, and we argue that future anti-fuzzing works should be designed in the scope of static techniques without self-modifying codes.

Future work. We consider that the future work of anti-fuzzing can focus on the following two aspects. On the one hand, we can keep reducing the overhead for existing anti-fuzzing techniques, which increases the number of defensive codes inserted into a program in a disguised way. A possible direction is that the anti-hybrid techniques in prior works are overqualified to disturb the program analysis. They use cryptography functions (e.g., hash, CRC) to wrap the variables. We have conducted some experiments, and it turns out that only hundreds of calculations are enough to overwhelm the symbolic executions, and they are cheaper than the heavy cryptography functions. On the other hand, future anti-fuzzing works can embed anti-fuzzing mechanisms into the program logic. For instance, similar to the flatten technique in obfuscation, we can split a block into several new blocks, and each new block contains a part of the assembly codes of the original block. Thus, each new block is logically dependent on the program and cannot be easily eliminated. Indeed, there are some challenges that need to be solved for this idea, e.g., how to maintain the context among different blocks and how to ensure the number of fake blocks is large enough.

6 Conclusion

In this paper, we design several practical and fully-automated anti-fuzzing techniques and integrate them into a prototype tool No-Fuzz. We optimize the storage cost of fake blocks as prior works insert them at the function level occupying an unrealistic storage room. In addition to the active anti-fuzzing techniques that disturb the fuzzing mechanisms, we also design the passive detection methods which precisely determine the running environments of the protected programs and launch mitigation techniques when binary-only-fuzzing exists. The evaluations demonstrate that No-Fuzz significantly reduces the branch coverage of fuzzers. Furthermore, we have also shown that No-Fuzz can impede bug findings in the LAVA-M dataset, i.e., fuzzers have to spend much more time finding a bug. We propose a new metric, the anti-fuzzing efficacy, to measure the defensive capability of an anti-fuzzing technique at a unit overhead cost. Based on this metric, we illustrate that No-Fuzz achieves the same or higher level of protection against fuzzers with even lower overhead than prior works.

In summary, we enhance the awareness of overhead and the importance of automation in an anti-fuzzing arms race. Inspired by this, we summarize the desired properties for future anti-fuzzing techniques - be with less overhead and automated. We have moved one step toward practical anti-fuzzing techniques and hope our efforts can further promote this topic.

7 Acknowledgements

This work was supported by the Research Grants Council of Hong Kong under Grants CityU 11217819, 11217620, 11218521, N_CityU139/21, RFS2122-1S04, C2004-21GF, R1012-21, and R6021-20F.

References

1. Google fuzzer test suite. https://github.com/google/fuzzer-test-suite. Accessed: 2022-03-12.
2. Qasan (qemu-addresssanitizer). https://github.com/andreafioraldi/qasan. Accessed: 2021-03-12.
3. A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2020-10-23, 2017.
4. Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. Announcing oss-fuzz: Continuous fuzzing for open source software. *Google Testing Blog*, 2016.
5. Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *Proc. of NDSS*, 2010.
6. Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proc. of ACSAC*, 2016.
7. Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proc. of ACM CCS*, 2016.
8. Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proc. of ACM CCS*, 2017.
9. Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proc. of IEEE ICSE*, 2022.
10. Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proc. of ACM CCS*, 2018.
11. Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: towards bug-driven hybrid testing. In *Proc. of IEEE S&P*, 2020.
12. Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *Proc. of IEEE S&P*, 2020.
13. B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *Proc. of IEEE S&P*, 2016.
14. Google. A scalable fuzzing infrastructure. https://github.com/google/clusterfuzz. Accessed: 2020-10-23.
15. Google. Syzkaller found bugs - linux kernel. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md.
16. Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. Antifuzz: Impeding fuzzing audits of binary executables. In *Proc. of USENIX Security*, 2019.

17. Munawar Hafiz and Ming Fang. Game of detections: how are security vulnerabilities discovered in the wild? In *Proc. of ACM ESE*, 2015.
18. Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. In *Proc. of ACM Measurement and Analysis of Computing Systems*, 2020.
19. Zhenghao Hu, Yu Hu, and Brendan Dolan-Gavitt. Chaff bugs: Deterring attackers by making software buggier. In *Arxiv*, 2018.
20. H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *Proc. of IEEE S&P*, 2020.
21. Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: Anti-fuzzing techniques. In *Proc. of USENIX Security*, 2019.
22. Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proc. of ACM workshop on Virtual machine security*, 2009.
23. Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proc. of ACM/IEEE ASE*, 2018.
24. Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Proc. of International Workshop on Recent Advances in Intrusion Detection*, 2011.
25. Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. In *Proc. of ACM Commun*, 1990.
26. Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *Proc. of International Conference on Information Security*, 2007.
27. Michael Rash. A collection of vulnerabilities discovered by the afl fuzzer. https://github.com/mrash/afl-cve Accessed: 2020-9-13.
28. Sanjay Rawat, Vivek Jain, Ashish Jith Sreejith Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proc. of NDSS*, 2017.
29. Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proc. of USENIX Security*, 2017.
30. Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proc. of NDSS*, 2016.
31. Robert Swiecki. Honggfuzz. 2020. https://github.com/google/honggfuzz.
32. D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *Proc. of IEEE S&P*, 2018.
33. T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proc. of IEEE S&P*, 2010.
34. Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear obfuscation to combat symbolic execution. In *Proc. of ESORICS*, 2011.
35. Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym : A practical concolic execution engine tailored for hybrid fuzzing. In *Proc. of USENIX Security*, 2018.
36. Michał Zalewski. American fuzzy lop. 2019. http://lcamtuf.coredump.cx/afl.
37. Michał Zalewski. Technical "whitepaper" for afl-fuzz. 2019. http://lcamtuf.coredump.cx/afl/technical_details.txt.

38. Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *Proc. of IEEE S&P*, 2021.

A Appendix

Table A1: The branch coverage of **A**NTI**F**UZZ and **F**U**Z**ZIFICATION

	readelf	objdump	size	nm	xmllint	libpng	gueutzli	sqlite3
AFL								
AF(cov)	1745	1449	839	706	1691	919	2103	3840
FZ(cov)	4498	2630	1247	1529	5556	1702	2805	4994
AFLFast								
AF(cov)	1913	1345	656	774	1503	958	2207	3757
FZ(cov)	3383	2848	1102	1411	5463	1492	2727	5065
Hfz-Q								
AF(cov)	1049	875	622	894	4117	1073	2499	4103
FZ(cov)	2422	2812	1277	1523	6122	1996	3040	6144
Hfz-P								
AF(cov)	1884	1067	899	632	3555	788	1931	3097
FZ(cov)	2612	2374	1148	1190	5312	1540	2563	4833
QSYM								
AF(cov)	1946	1276	814	932	4255	1287	2493	4441
FZ(cov)	5072	3731	1448	1740	6442	2084	3538	7433

Table A2: The overhead comparisons between upx and existing anti-fuzzing techniques. As emphasized in the table, the overhead of existing anti-fuzzing techniques is more or less close to that of packing techniques.

	readelf	objdump
Exec time	125.4 ms	2156.2 ms
upx	**+13.0%**	+24.3%
AF(cov)	+37.7%	**+2.0%**
FZ(cov)	+32.9%	+14.6%
Storage cost	3.22 M	9.94 M
upx	**-2.24M(-69.6%)**	**-7.44 M (-74.8%)**
AF(cov)	21.27 M (+660.6%)	21.25 M (+213.8%)
FZ(cov)	1.29 M (+40.1%)	1.28 M (+12.9%)

Table A3: Real-world programs of different size and execution time.

size & exec time	files
Small	readelf, objdump, size, nm guetzli, libpng, sqlite3, xmllint
Large	ffmpeg_g, nomacs, calc, impress

Table A4: We fuzz a dummy program as well as the obfuscated versions. The dummy program contains several magic-byte checks and will crash if the constraints are satisfied. Tigrees(S) only applies static obfuscation, while Tigrees(D) adopts self-modifying codes which are dynamic.

	First Crash	Fuzz Rate
Native	45s	1805 exc/s
UPX	1m27s	1800 exc/s
Tigress(D)	$+\infty$	0 exc/s
Tigrees(S)	2m4s	1667 exc/s
llvm-obfuscator	1m48s	1400 exc/s

Table A5: Evaluations to launch BOF on obfuscated/packed binaries. $\times$ means BOF cannot initialize on the binary within 30 minutes for all four fuzzers. $\surd$ means all fuzzers succeed in launching the BOF. As the table suggests, self-modifying codes (upx & obfuscation with JIT) can completely prevent the BOF from initialization.

	native	upx	llvm-obf	Tigress(D)	Tigress(S)
dummy	$\surd$	$\surd$	$\surd$	$\times$	$\surd$
binutils	$\surd$	$\times$	$\surd$	$\times$	$\surd$
libjpeg	$\surd$	$\times$	$\surd$	$\times$	$\surd$
libpng	$\surd$	$\times$	$\surd$	$\times$	$\surd$
libtiff	$\surd$	$\times$	$\surd$	$\times$	$\surd$
ffmpeg	$\surd$	$\times$	$\surd$	$\times$	$\surd$
gzip	$\surd$	$\times$	$\surd$	$\times$	$\surd$