Practical Anti-Fuzzing Techniques With Performance Optimization

ZHOU, Zhengxiang; WANG, Cong

Published in:
IEEE Open Journal of the Computer Society

Published: 01/01/2023

Document Version:
Final Published version, also known as Publisher’s PDF, Publisher’s Final version or Version of Record

License:
CC BY-NC-ND

Publication record in CityU Scholars:
Go to record

Published version (DOI):
10.1109/OJCS.2023.3301883

Publication details:
ZHOU, Z., & WANG, C. (2023). Practical Anti-Fuzzing Techniques With Performance Optimization. *IEEE Open Journal of the Computer Society*, *4*, 206-217. https://doi.org/10.1109/OJCS.2023.3301883

IEEE Open Journal of the
Computer Society

Received 8 July 2023; accepted 1 August 2023. Date of publication 4 August 2023;
date of current version 22 August 2023. The review of this article was arranged by Associate Editor Quan Chen.

Digital Object Identifier 10.1109/OJCS.2023.3301883

Practical Anti-Fuzzing Techniques With Performance Optimization

ZHENGXIANG ZHOU AND CONG WANG (Fellow, IEEE)

Department of Computer Science, City University of Hong Kong, 518057, Hong Kong

CORRESPONDING AUTHOR: CONG WANG (e-mail: congwang@cityu.edu.hk).

This work was supported by the Research Grants Council of Hong Kong under Grants CityU 11217620, 11218521, N_CityU139/21, RFS2122-1S04, C2004-21G, C1029-22G, R1012-21, and R6021-20F.

ABSTRACT Fuzzing, an automated software testing technique, has achieved remarkable success in recent years, aiding developers in identifying vulnerabilities. However, fuzzing can also be exploited by attackers to discover zero-day vulnerabilities. To counter this threat, researchers have proposed anti-fuzzing techniques, which aim to impede the fuzzing process by slowing the program down, providing misleading coverage feedback, and complicating data flow, etc. Unfortunately, current anti-fuzzing approaches primarily focus on enhancing defensive capabilities while underestimating the associated overhead and manual efforts required. In our paper, we present No-Fuzz, an efficient and practical anti-fuzzing technique. No-Fuzz stands out in binary-only fuzzing by accurately determining running environments, effectively reducing unnecessary fake block overhead, and replacing resource-intensive functions with lightweight arithmetic operations in anti-hybrid techniques. We have implemented a prototype of No-Fuzz and conducted evaluations to compare its performance against existing approaches. Our evaluations demonstrate that No-Fuzz introduces minimal performance overhead, accounting for less than 10% of the storage cost for a single fake block. Moreover, it achieves a significant 92.2% reduction in total storage costs compared to prior works for an equivalent number of branch reductions. By emphasizing practicality, our study sheds light on improving anti-fuzzing techniques for real-world deployment.

INDEX TERMS Anti-fuzzing, fuzzing, software engineering, software protection.

I. INTRODUCTION

Fuzzing, a software testing technique introduced in 1990 [29], involves supplying a target program with randomly generated inputs to detect program bugs by observing abnormalities such as segmentation faults. In recent years, fuzzers have undergone significant evolution, with researchers incorporating techniques such as program instrumentation [2], [11], [32], [33], [35], [41], [43] and program analysis [34], [37], [40] to enhance bug-finding efficiency. Additionally, there have been explorations of modifications to classic fuzzing mechanisms, such as resource reallocation for specific tasks [5], [6], [8], [9], [22], [25], [46]. These advancements have resulted in significant successes in uncovering numerous bugs [15], [16], [31], [35].

However, exposing bugs in a program is a double-edged sword. On one hand, developers can identify and resolve bugs before they propagate across the internet. On the other hand, attackers can also exploit fuzzers to discover zero-day vulnerabilities, leading to potential financial losses for companies.

While adversaries can manually analyze commercial software, recent studies [18], [36] have demonstrated that attackers are increasingly inclined to employ automated tools like fuzzers to identify vulnerabilities, rather than relying solely on manual analysis. In response to the growing challenge of bug discovery, anti-fuzzing techniques have been proposed to impede the malicious use of fuzzers, such as ANTIFUZZ [17] and FUZZIFICATION [23].

The objective of anti-fuzzing is to maintain the advantageous position of developers in bug-finding. These techniques introduce penalties to disrupt fuzzing heuristics or slow down the rate of fuzzing. The source code of the protected program is compiled into two versions: one with anti-fuzzing code incorporated and the other remaining unmodified. Developers retain the original version for thorough testing, while adversaries can only access the protected version, where the anti-fuzzing code significantly hinders the effectiveness of fuzzers. Consequently, developers are expected to discover a significantly larger number of bugs compared to adversaries and address them promptly to mitigate potential losses stemming from zero-day vulnerabilities.

While anti-fuzzing techniques show promise, there is room for improvement in the prototypes presented in previous works to achieve a more fine-grained application scope. Consideration of storage overhead is crucial for the practical adoption of anti-fuzzing techniques.

In previous works, the insertion of fake blocks into the program was used to saturate the bitmap of fuzzers. However, this approach can significantly increase the program's size, sometimes even several times larger than the original program. As a result, developers may be reluctant to bear such storage costs solely for anti-fuzzing purposes. Instead, they may opt for lighter obfuscation tools whenever possible, despite the fact that these tools may not offer sufficient protection against fuzzers.

Another important factor is the automation of the tools. Existing prototypes require manual identification of specific code areas, and they may have dependencies on third-party tools or libraries that could be incompatible with users' operating systems. These factors pose challenges to the future design and implementation of anti-fuzzing techniques. Ideally, anti-fuzzing techniques should possess the following two properties:

P1) *Minimization of both storage and performance overheads.*
P2) *Support for automation without modifying the development procedures of the program.*

Based on these considerations, we propose our solution to anti-fuzzing techniques, which consists of two categories: passive detection methods and active disturbance methods. The passive detection methods are responsible for precisely monitoring whether the protected program is being subjected to fuzzing and implementing mitigation strategies once fuzzers are detected. Our design incorporates instrumentation checking and execution frequency checking to achieve anti-fuzzing techniques with minimal overhead.

The active disturbance methods, on the other hand, focus on impeding fuzzers by attacking their underlying assumptions and disrupting their normal operation. To optimize their effectiveness, we refine the defective fake blocks and employ anti-hybrid techniques. In our design, the storage overhead associated with the fake blocks is minimized, accounting for less than 10% of the overhead reported in previous works. Additionally, we replace the cryptography functions utilized in prior approaches with lightweight arithmetic functions. This substitution enables the automatic insertion of protections into the majority of program areas.

We have implemented these techniques in the form of a fully automated tool called No-Fuzz. No-Fuzz seamlessly incorporates the anti-fuzzing techniques directly into the source code of programs, eliminating the need for modifications to the compilation procedures such as header files, linked libraries, or compilation commands. This design ensures a straightforward integration process without disrupting the existing development workflows. Furthermore, it is important to note that No-Fuzz is compatible with other anti-fuzzing techniques from previous works.

In our evaluation, we assess the effectiveness of our techniques in reducing branch coverage using real-world software from Binutils and two popular benchmarks: Google FTS [1] and Magma [19]. With all techniques enabled, No-Fuzz achieves an average reduction of 59.6% in branch coverage. Furthermore, we demonstrate the ability of our techniques to prevent bug findings using the LAVA-M dataset [12].

To validate our optimizations compared to prior works, we conduct a comparison between No-Fuzz and the corresponding techniques in ANTIFUZZ and FUZZIFICATION . The results reveal that our design introduces less overhead and effectively mitigates the negative impact of anti-fuzzing techniques on regular users. Specifically, we achieve a reduction of approximately 92.2% in storage cost compared to the prior works for the same number of branch reductions.

Moreover, we address the challenge posed by the lack of a suitable metric for comparing different anti-fuzzing techniques. Existing approaches typically measure the anti-fuzzing effects and overhead separately. However, performance and overhead are orthogonal factors that vary based on different configurations. It is unfair to directly compare the performance of different works with unequal overhead. Therefore, in addition to measuring performance and overhead separately, we propose a novel metric called "anti-fuzzing efficacy". This metric establishes a link between the two metrics, enabling the measurement of increased defensive capability per unit overhead.

In summary, this paper contributes in the following ways:
1) It sheds light on the shortcomings of existing anti-fuzzing prototypes and outlines the desirable properties of ideal anti-fuzzing techniques.
2) It designs and implements an automated anti-fuzzing prototype called No-Fuzz, which is capable of detecting and disrupting run-time fuzzing mechanisms.
3) It evaluates No-Fuzz and several prior anti-fuzzing techniques using common benchmarks, demonstrating No-Fuzz's minimal impact on the protected binary and its effectiveness in impeding binary-only fuzzing.

It is important to note that a conference version of this work was previously published [45], which did not investigate anti-hybrid techniques. This extended work introduces two novel lightweight anti-hybrid techniques that facilitate the practical deployment of anti-fuzzing techniques. Additionally, a more extensive set of experiments is conducted to assess the effectiveness of No-Fuzz.

The source code for all the implemented tools is available at https://github.com/CongGroup/No-Fuzz.

II. TECHNICAL BACKGROUND OF ANTI-FUZZING

The objective of anti-fuzzing techniques is to counteract fuzzers and reduce the number of reported bugs in protected binaries. These techniques can be broadly classified into three categories: anti-fast-execution, anti-feedback, and anti-hybrid techniques, based on their impact on different fuzzing mechanisms. We will provide a brief overview of these techniques in the following sections.

Anti-fast-execution: Introducing latency to the binary: One of the underlying assumptions of fuzzers is that exploring more paths in the binary can be achieved by executing a large number of trials with different inputs. Fuzzers are typically designed with acceleration techniques that feed thousands of seeds per second into the program under test (PUT) [42]. Anti-fast-execution techniques aim to disrupt this fast execution by introducing latency into the binary.

EscapeFuzz [13] proposes reducing the maximum number of executions per second as a means to introduce latency. However, a major challenge lies in ensuring that the latency does not adversely affect regular users. ANTIFUZZ [17] manually inserts delay functions in error handling code, while FUZZIFICATION [23] introduces latency functions in cold blocks. Both techniques attempt to introduce delays in areas that are seldom reached by regular users but are susceptible to triggering fuzzers.

Anti-feedback: Disturbing the feedback information: Modern fuzzers heavily rely on two types of feedback to guide their fuzzing heuristics: coverage-feedback and error signals. The coverage information is stored in a bitmap of limited size, and fuzzers make decisions on seeds and mutations to maximize the coverage. The error signals indicate to fuzzers the seeds that trigger bugs, which is the ultimate goal of using fuzzers. Anti-feedback techniques aim to disrupt the feedback mechanisms by inserting fake blocks into the protected binary. These blocks contain code that is irrelevant to the program logic but is recorded as valid blocks in the coverage bitmap. By occupying a significant portion of the bitmap with these fake blocks, fuzzers are unable to update new coverage information.

ANTIFUZZ and VALL-NUT [26] redirect control flow to randomly generated fake functions within the protected program. FUZZIFICATION inserts a fixed number of constraints and functions into the binary, creating ROP chains as fake paths within assembly code snippets. Additionally, FennecFuzz [10] modifies the fake blocks to specifically hinder a different coverage metric called "coverage counting," which is only used by TortoiseFuzz [38].

Regarding error signals, ChaffBugs [20] suggests inserting non-exploitable bugs into the binary to confuse the segmentation faults reported to fuzzers. ANTIFUZZ proposes an approach to hinder crash discovery by installing a signal handler. The handler conceals signals from fuzzers by elegantly terminating the program, thereby preventing the detection of crashes.

Anti-hybrid. Impeding program analysis: Hybrid fuzzers [9], [21], [34], [40] heavily rely on techniques such as taint analysis and symbolic execution to accelerate the fuzzing process.

Anti-hybrid techniques aim to disrupt these program analysis techniques by introducing complex data flows into the protected binary. The underlying idea is that program analysis techniques struggle to handle intricate data flows due to limited CPU resources.

ANTIFUZZ encrypts and decrypts the inputs and transforms variables in critical comparisons into their hash values. Similarly, FUZZIFICATION adds additional copy operations to the operand string, complicating the data flows and misleading taint analysis engines by providing incorrect tag maps.

III. NO-FUZZ DESIGN

A. OVERVIEW OF NOFUZZ

No-Fuzz incorporates both passive detection methods and optimized active techniques from previous works, including fake blocks and anti-hybrid techniques. Passive detection methods are employed to identify whether the protected binary is being subjected to binary-only fuzzing (BOF). Upon detecting the presence of fuzzers, the protection triggers mitigation mechanisms, such as introducing latency to impede fast execution.

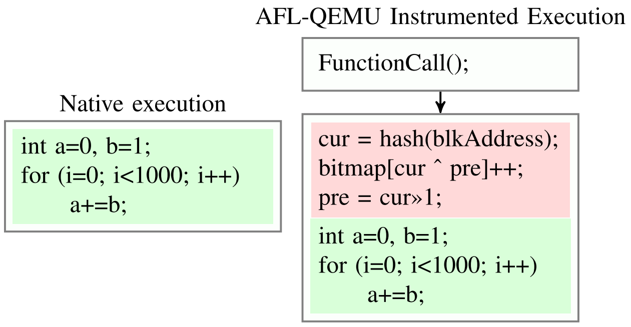
As mentioned earlier, the active techniques employed in previous works face practical limitations due to storage overhead. To overcome this challenge, we have implemented two key optimizations. Firstly, we optimize the fake blocks and design the landing space by leveraging the block-identification mechanism of binary-only instrumentation. This optimization significantly reduces the storage overhead of a fake block to just one byte.

Secondly, we replace the heavyweight cryptography functions that were used in previous anti-hybrid techniques with lightweight infinite series. By employing this approach, we achieve similar functionality with reduced computational complexity. Additionally, we introduce the concept of an input-tainted constant. This concept hampers symbolic execution by increasing the number of unsolvable variables in constraints. These variables possess invariant values, which have no impact on the original execution.

B. PASSIVE DETECTION METHODS

A crucial aspect of anti-fuzzing techniques is to ensure that the inserted measures do not adversely affect regular users. It is essential to minimize the impact on normal program execution while effectively countering fuzzing attempts. In an ideal scenario, the protected program should have the capability to accurately detect the presence of fuzzers, enabling us to covertly impose severe penalties on adversaries. To achieve this objective, we introduce passive detection methods that allow us to identify the execution environments of protected binaries. Once fuzzers are detected, we can activate mitigation mechanisms such as delaying the execution or aborting the program altogether to prevent successful fuzzing attempts.

Method 1. Detect binary-only instrumentation: In the context of utilizing anti-fuzzing techniques, adversaries are unable to access the source code of the protected binary, as they solely rely on the binary-only mode of fuzzers. However, regardless of the specific techniques they employ, adversaries must gather coverage information from the target program.



```
FunctionCall();

cur = hash(blkAddress);
bitmap[cur ^ pre]++;
pre = cur»1;

int a=0, b=1;
for (i=0; i<1000; i++)
    a+=b;
```

Native execution:

```
int a=0, b=1;
for (i=0; i<1000; i++)
    a+=b;
```

FIGURE 1. Instrumentation to collect coverage feedback.

Various methods, such as dynamic instrumentation, hardware assistance, and binary rewriting, are commonly used to achieve this observation. It's important to note that all coverage collection mechanisms introduce a noticeable latency, creating a timing gap in the program under test (PUT) compared to static instrumentation.

Fig. 1 illustrates the coverage mechanism of AFL-QEMU, where the additional code within the red block introduces additional performance overhead for a function-call branch. By detecting the timing gap between native execution and execution with coverage-collecting code, we can determine the running environment.

Timing-related techniques are widely used in the field of malware detection [3], [24], [28]. Taking inspiration from existing research, we have developed a detection mechanism based on binary instrumentations.

In the native execution environment (real CPU), control flow proceeds directly to the block following a branch-taken instruction. However, in the case of binary-only fuzzing (BOF), the instrumented program executes additional instructions at the beginning of a block to collect coverage data. To detect BOF, we examine the edge instructions count (EIC), which represents the estimated number of instructions executed when entering a function or a block (i.e., instructions of an edge). Through experiments, we have observed that the EIC for BOF can be approximately ten times larger than that in native execution. By detecting this timing gap in a protected program, we can identify the presence of BOF and proceed with appropriate mitigation techniques.

Mitigation. Introduce latency: It is important to address the issue of false positives in the detection process, as the performance of CPUs can vary, leading to certain executions having a relatively large timing gap even in the native environment. This discrepancy often occurs during CPU context switching, where the additional overhead is included in the timing gap measurement. In our experiments, we found that 0.03% of executions resulted in false positives in a stable environment, while the false-positive rate increased to 0.1% in a busy environment with a high number of parallel tasks being executed simultaneously.

To ensure that false positives do not adversely affect regular users, the mitigation mechanisms for fuzzing should be implemented moderately. In our approach, we introduce a one-second latency to the program by triggering IO blocking when the BOF instrumentation is detected. Although one second may not be sufficient to significantly impact a fuzzer, the overall effectiveness of the penalty can be guaranteed by inserting multiple detection functions into the protected program.

Method 2. Examining execution frequency: The nature of fuzzing entails repeated executions of PUT within a short period of time. This characteristic can be utilized to detect whether the program is being fuzzed. Inspired by the proverb "many a little makes a mickle," we propose that if the PUT leaves traces or vestiges after each fuzzing round, these vestiges will accumulate during the rapid re-executions. Over time, they will grow significant enough to indicate to PUT that it is being fuzzed.

In our design, the protected program creates a temporary file with each run. By monitoring the rate of file creation, we can determine if a fuzzer is present. Specifically, if more than 60 files are created within a minute (the threshold is configurable), the program is alerted to the presence of a fuzzer. However, managing these temporary files poses a challenge. Traversing and identifying the files created by the protected program can be time-consuming, and failing to delete them can disrupt the file system for regular users.

To address this challenge, we leverage a daemon process for the management of temporary files. A daemon process runs as a background process, detaching itself from the parent process and continuing to run even after the parent process terminates. In our design, the daemon process acts as a patrolling agent for the temporary files. It prevents unintentional deletion of the files and deletes them after patrolling.

During execution, the patrolling daemon process detaches itself from the protected program and creates temporary files with sequential IDs to indicate their order. These files are created in ascending order, with the largest order representing the detection threshold for fuzzers. The daemon process then locks the file for a specified period, which we refer to as the "patrolling time." After the patrolling period, it checks if the locked file is correct and deletes the file it created.

Meanwhile, the protected program searches for the temporary files with the threshold order during each execution. Once the file is found, it indicates that the program has been executed more than the threshold number of times within the patrolling time, suggesting the likely presence of BOF. At this point, appropriate mitigation techniques can be applied.

Mitigation. Aborting program: In contrast to the timing gap approach, the results obtained from the daemon process are highly accurate, with no false positives. This allows us to implement a more severe penalty in this mitigation method. The PUT has the capability to either abort the execution or trigger an artificially inserted bug, thereby misleading fuzzers about the occurrence of crashes. To further ensure that the mitigation strategy does not adversely affect regular users in unexpected scenarios, developers can configure a longer patrolling time (e.g., 5 minutes) and a larger threshold (e.g., 1000 files). Given

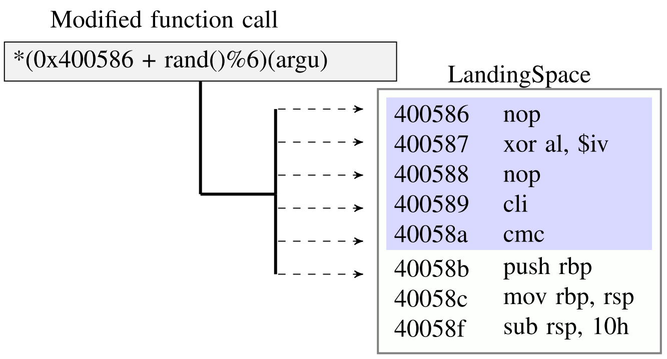


FIGURE 2. Function with landing space.\$iv is the immediate value, in this example, it will be 0x90 which is the opcode for nop.

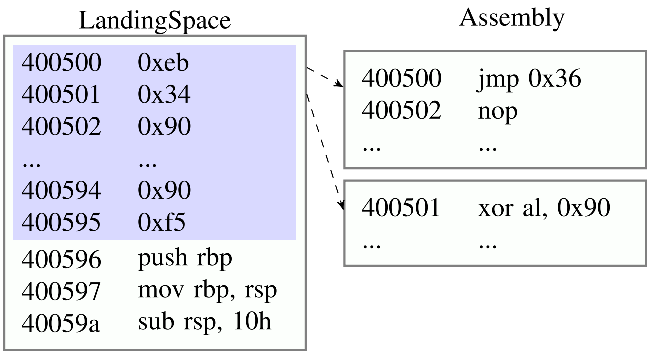


FIGURE 3. Jump over unnecessary blocks.

the infrequency at which regular users typically execute the program with such high frequency, the likelihood of regular users being affected is minimal.

C. ACTIVE METHODS: MINIMUM FAKE BLOCKS

Existing fake blocks incur a non-optimal storage cost, posing challenges for small programs that are sensitive to high storage overhead. Attackers often target small programs due to their faster execution speed and less complex logic. As fuzzers evolve and enhance their capabilities, such as larger bitmaps and improved heuristics, anti-fuzzing techniques must adapt by inserting more protection code, leading to unsustainable storage overhead in the arms race.

To minimize the extra storage overhead of fake blocks, we focus on the assembly level and identify unnecessary code that contributes to the overhead. For instance, C compilers generate function frames that are unrelated to anti-fuzzing and can be eliminated to reduce storage costs.

Attackers rely on binary-only fuzzing (BOF) to collect coverage feedback without access to source code. These tools insert code before entering a new block and generate new block records when encountering control-flow-changing instructions. By instrumenting each function with a code segment called the "landing space," consisting of instructions that have no impact on normal execution, we ensure that each byte in the landing space can be translated into a valid instruction. We also modify the destination address of function calls to random bytes in the landing space. When invoked, the control flow "lands" at a random instruction, triggering the fuzzer to record new coverage. Over multiple fuzzing rounds, the fuzzer records a significant number of possible addresses in the landing space, overwhelming its bitmap with corresponding fake coverage. This approach reduces the storage cost of fake blocks to a minimum, which is around 10% of prior works.

Fig. 2 displays the assembly code of a function and the corresponding landing space. The original destination address of the function is *0x40058b*, and we insert the landing space before this address in the text section, occupying the memory region between *0x400586* and *0x40058a*. By modifying the function call (*0x400586* + *rand()%6*), the control flow can either jump to the landing space or the original start of the function. In this example, a fuzzer will record six fake blocks, incurring a minimal cost of six bytes.

Although the initial implementation of the landing space appears to disrupt the coverage feedback of binary-only fuzzing (BOF), we have identified certain limitations. Excessive size of the landing space introduces noticeable latency to the protected program. Moreover, the addresses of fake blocks within the landing space are closely packed, leading to higher chances of hash collisions during fuzzing. A higher frequency of hash collisions reduces the saturation of the fuzzer's bitmap with fake blocks, allowing it to better identify genuine branches in the protected program. To address these limitations, we propose two optimizations.

Optimization 1. Jump over unnecessary instructions: When the control flow reaches the initial bytes of the landing space, it must execute all subsequent instructions, resulting in significant latency for larger landing spaces. To avoid this unnecessary execution, we modify certain one-byte instructions to short jumps, with the jump offset being the opcode of the following instruction. As depicted in Fig. 3, if the control flow lands at address *0x400500*, the corresponding assembly code is transformed into a two-byte short jump with an offset of *0x36*. On the other hand, if it lands at the subsequent byte *0x400501*, the assembly code becomes "xor al, 0x90". Despite these modifications, the landing space retains its functionality as every byte can still be correctly disassembled and recorded as a new block. By introducing these jump instructions, the performance overhead is reduced to about 5% for a landing space of 1000 bytes.

Optimization 2. Spraying LandingSpace at different addresses: To decrease the occurrence of hash collisions, we introduce multiple landing spaces distributed at different addresses. This is achieved by encapsulating functions in the original program with intermediate blocks. As shown in Fig. 4, these intermediate blocks solely redirect the control flow within the protected binaries without impacting the program execution. By intentionally creating significant address disparities among the intermediate blocks, we increase the likelihood of generating diverse hash values compared to using a single landing space. Consequently, the size of the

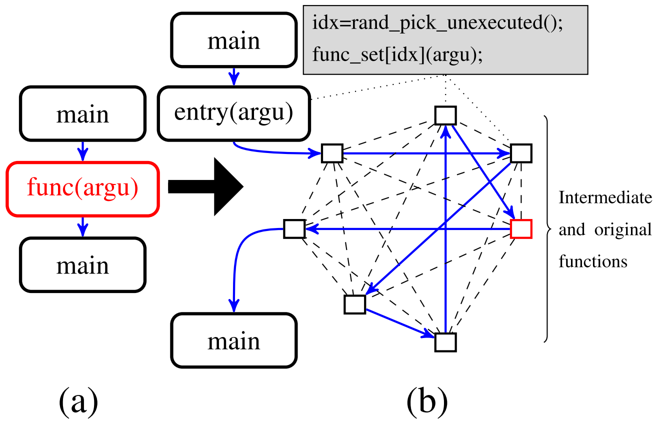


FIGURE 4. Spray intermediate functions in protected binaries.

landing spaces within these functions can be reduced, as they are now distributed among the intermediate blocks.

D. ACTIVE METHODS: ANTI-HYBRID THROUGH LIGHTWEIGHT ARITHMETIC FUNCTIONS

Existing anti-fuzzing techniques against hybrid fuzzers are overly restrictive. Previous works [17], [23] have suggested complicating constraints and disrupting input taint analysis as effective methods to prevent symbolic execution. State-of-the-art techniques employ heavyweight cryptographic functions, such as hash functions, to encapsulate inputs and critical comparisons. However, these functions produce unordered values and are only utilized in equality comparisons. Moreover, the substantial overhead they introduce becomes significant when applied to a large number of functions. To disrupt taint analysis, additional data flows are introduced to confuse input taint analysis, such as duplicating the original string into a temporary string array. Similar to heavyweight wrapping functions, the overhead from these implicit data flows accumulates as the number of conditions requiring protection increases. Due to these limitations, these designs are only employed in critical areas, necessitating manual effort from developers. To promote fully automated defensive techniques, lightweight functions and more flexible approaches are preferred in anti-hybrid-fuzzing techniques.

Lightweight and complex constraints: To address these challenges, we propose VariableMapping, which overwhelms symbolic execution with lighter and comparable self-mapping functions. While hash functions or CRC functions used in existing approaches are complex and difficult to reverse, a determined attacker can analyze a legal input or disassemble the binary to obtain the correct values for hash/CRC comparisons. Consequently, applying such functions is unnecessary, as their complexity only provides limited defense improvement. Instead, we utilize infinite sum functions that output the input when the number of items is sufficiently large. VariableMapping employs lighter and comparable self-mapping functions to overwhelm symbolic execution. These functions are much smaller than hash functions and can be controlled by adjusting the number of items. Moreover, they effectively impede symbolic execution engines due to the involvement of numerous floating-point and approximate calculations.

There are numerous alternative mapping functions that can be applied together in a protected program. We select the Maclaurin series of $\frac{1}{1-x}$ due to its difficulty and controllability. The function is defined as $f(x) = \frac{1}{1-x} = \sum_{i=0}^{N} x^i$ (where $|x| < 1$ and N is sufficiently large). This function incorporates multiple power calculations that are difficult to solve, and its complexity scales linearly with the number of items, allowing for better latency control. Furthermore, the Maclaurin series is relatively lighter than hash functions, making it suitable for automatic application within VariableMapping. We perform a simple static taint analysis to identify all variables that directly access the input bytes. Each variable, denoted as var, is replaced with the mapping function $f(1 - \frac{1}{var}) = var$, which outputs the value var. This mapping can also be applied to strings by mapping characters to integers. The overhead introduced by a single mapping function is negligible, making the overall overhead acceptable for covering most input-related variables.

Another advantage of VariableMapping is that the results of the mapping functions maintain order. Unlike hash functions that cannot preserve input order, where larger inputs do not necessarily result in larger hash values, VariableMapping returns original values that are compatible with various types of comparisons, including equations and inequalities.

Input-tainted constants: Symbolic execution aims to determine feasible values for input-tainted variables based on the given constraints. Solving these constraints becomes more challenging as the number of variables increases. Leveraging this insight, we introduce input-tainted constants, which are invariant values derived from input bytes. One such constant is the Kaprekar constant, but there are other alternatives to consider for future defenses against evolving tools. The Kaprekar constant is obtained by applying a series of arithmetic operations to any four-digit number, eventually converging to the value 6174. This constant serves as a target that symbolic execution cannot execute, effectively exploiting a weakness in the approach. From the perspective of symbolic execution, the Kaprekar constant behaves as an infinite loop function. To create input-tainted constants, we randomly select some input bytes and assemble them into a four-digit number, such as adding them together and getting the modulo by dividing 10000. These variables are transformed into the Kaprekar constant and converge to 6174, becoming the input-tainted constants. We can incorporate these constants into any constraints within the protected program. Since calculating the Kaprekar constant converges after at most seven iterations of the function, the overhead is negligible compared to existing techniques that disrupt input-taint analysis. Furthermore, the input-tainted constants behave as constant values, allowing us to add them to constraints by multiplying the constraints with $Kar/6174$, without affecting the original constraints.

```
void main(){
   int kar;
   char input[10];
   FILE* fp = fopen("input", "r");
   fread(input, sizeof(char), 8, fp);
// Assemble some input bytes into input
    -tainted constant 'kar' and the
    value is 6174
   kar = Kaprekar6174(input[3]+input
      [4]<<8);

// Disturb taint analysis and add a new
     variable to the constraint
   if (input[0]*kar/6174 == 'a')

// Variable mapping that outputs
    ordered values
// It optimizes the hash comparisons
// if(hash(input[1]) == 'hash value')
   if (varmap(1-1/input[1]) > 'b')

// Combine them together
   if (varmap(1-1/input[5]) > 'c'*kar
      /6174){
      // do something
   }
}
```

FIGURE 5. Example of Nofuzz anti-hybrid techniques.

Fig. 5 demonstrates an example of the Nofuzz anti-hybrid technique. The input variable *input[1]* is transformed into an infinite sum using the function *varmap(1-input[1])*, producing the same output as the original *input[1]*. Additionally, we can multiply variables with the input-tainted constant *kar*. The value of *kar* remains invariantly 6174, according to the properties of the Kaprekar constant, while it can be derived from random input bytes (such as *input[3]* and *input[4]* in the example). Both VariableMapping and input-tainted constants effectively impede symbolic execution while exerting minimal impact on the execution of the protected program.

IV. EVALUATION

We conduct an evaluation of No-Fuzz to address the following research questions (**RQ**s):

- *RQ 1:* Can No-Fuzz effectively impede fuzzers from exploring new branches?
- *RQ 2:* How successful are the anti-fuzzing techniques in preventing fuzzers from discovering bugs?
- *RQ 3:* What is the storage and performance overhead involved in deploying anti-fuzzing techniques?
- *RQ 4:* Which metric is suitable for comparing different anti-fuzzing techniques?

To address **RQ 1**, we consider that coverage is independent of the bug-finding capabilities of fuzzers [7]. A fuzzer's ability to cover more code paths increases the likelihood of finding bugs within the target program. We evaluate the reduction in coverage on real-world binaries after applying No-Fuzz, demonstrating the effectiveness of anti-fuzzing techniques as a defense mechanism.

To investigate **RQ 2**, we evaluate the anti-fuzzing techniques using the LAVA-M benchmark [12], measuring the shortest time required to discover a bug. The benchmark includes four flawed binaries (`base64`, `md5sum`, `who`, `uniq`) with artificially inserted bugs ranging from dozens to thousands.

Both **RQ 3** and **RQ 4** pertain to the evaluation of overhead associated with anti-fuzzing techniques. For **RQ 3**, we assess the storage and performance overhead on real-world programs of varying sizes. Regarding **RQ 4**, we address concerns that the overhead alone is insufficient for the comprehensive evaluation of anti-fuzzing techniques. Increasing the amount of defensive code added to the protected programs is likely to increase the overhead. The defensive capability of a technique is distinct from the additional overhead, as the more defensive code is inserted into the protected binary, the safer it becomes. Thus, a single metric (anti-fuzzing effect or overhead) is inadequate for assessing a defensive technique. We propose combining these two metrics and unifying the evaluation criteria by measuring the defensive ability relative to the unit cost of storage or execution rate. We introduce a new metric called *anti-fuzzing efficacy*, which measures the number of reduced branches per byte of additional storage cost and per millisecond of latency. This metric evaluates the capability of anti-fuzzing techniques in relation to the introduced overhead.

Throughout the experiments, the latency mitigation is set to one second, with the daemon process patrolling for one minute and issuing an alert if the number of executions exceeds 60 times. The landing space is configured to occupy 100 bytes, and the functions are wrapped in 50 intermediate functions. AFL and AFL-based fuzzers (AFLFast and QSYM) utilize AFL-QEMU. HonggFuzz supports both Intel-PT and QEMU, and we employ binary-only modes for both. Each fuzzing campaign runs on three CPU cores. Notably, QSYM operates two AFL instances with two CPU cores and an SMT solver using one core. Fuzzing campaigns on the LAVA-M dataset run for 48 hours, while others run for 24 hours. Due to the non-deterministic nature of fuzzing behaviors, we repeat each fuzzer x target combination ten times.

A. REDUCING CODE COVERAGE

We assessed the branch coverage of five fuzzers on 12 real-world binaries obtained from Binutils, Magma, and Google FTS. Fig. 6 displays the average number of covered branches by each fuzzer, both with and without No-Fuzz protection. We conducted separate evaluations for each technique to avoid one technique masking the effects of others. The combined application of all No-Fuzz techniques significantly impedes the branch exploration of fuzzers. On average, fuzzers can only discover 40.4% of the expected branches, most of which pertain to initializations and input correctness checks.

A single passive detection technique reduces branch coverage by 19.7% to 85.1%. The variation in effectiveness is influenced by the choice of mitigation techniques and the

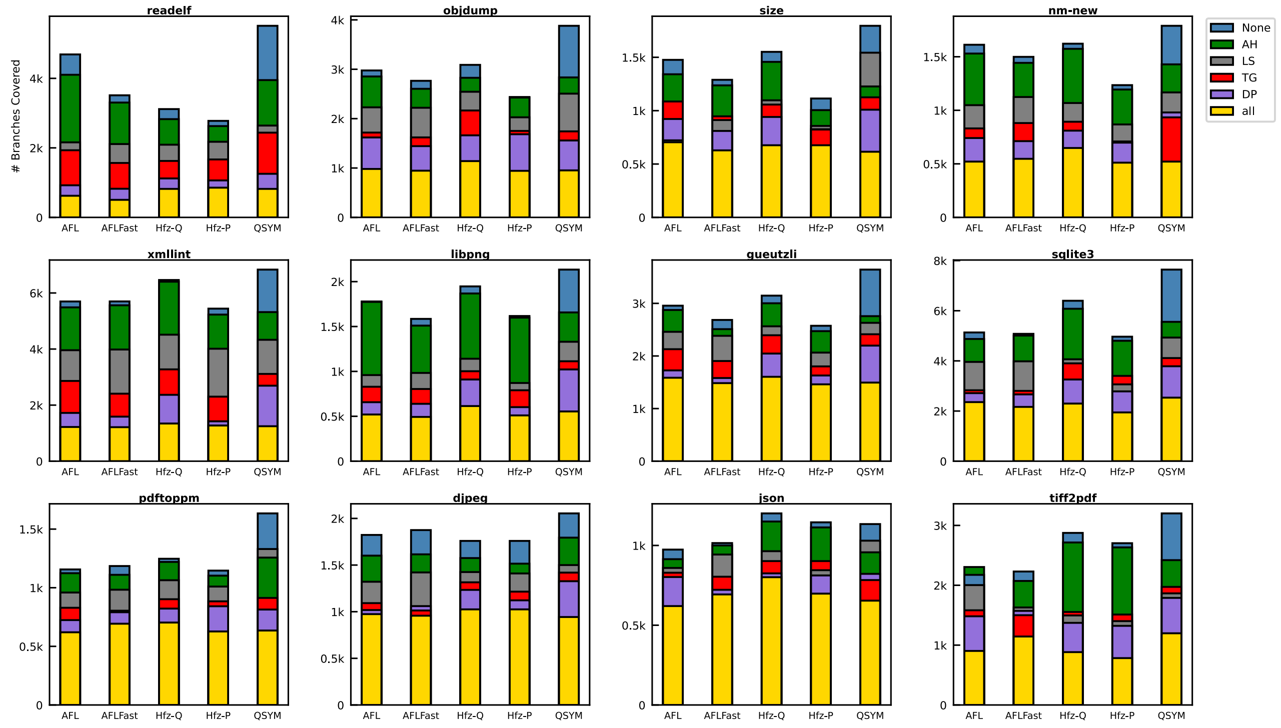


FIGURE 6. **Branch covered by four fuzzers against twelve binaries with and without different protections. The techniques are Timing Gap, Daemon Process, Landing Space, and Anti-Hybrid. The fuzzers are AFL, AFLFast, Honggfuzz-QEMU,Honggfuzz-PT, and QSYM .**

differences among fuzzers. As depicted in the results, aborting the PUT (represented by the purple column) is more effective than introducing latency (represented by the gray columns) to the protected programs. However, introducing latency has a slightly lesser impact on users compared to program abortion. This trade-off between effectiveness and user impact prevents us from concluding that one mitigation technique outperforms the others. Our recommendation is to employ more severe mitigation techniques with more precise detection techniques.

The landing space obstructs an average of 27.9% of the branches. It is less effective than passive detection methods because the penalty for fuzzers is limited to bitmap saturation, allowing fuzzers to continue running normally. Anti-hybrid techniques exhibit effectiveness against QSYM, reducing coverage by 23.2%. However, their impact on other fuzzers is limited. The reduction in coverage of traditional fuzzers against anti-hybrid techniques can be attributed to the latency introduced by the protection.

B. PREVENTING FUZZERS FROM FINDING BUGS

LAVA-M benchmark: Despite recent works suggesting the use of up-to-date benchmarks such as the Google fuzzer test suite [1], Magma [19], UNIFUZZ [27], and FIXREVERTER [44] for bug-finding experiments, we find them unsuitable for evaluating BOF techniques. The issue arises because some of these benchmarks rely on sanitizers, which most BOF techniques do not support. While there are works like QEMU-AddressSanitizer [14] that try to facilitate this field, our evaluation covers different BOF techniques, not all of which have complementary tools. Furthermore, these new benchmarks pose higher difficulty in bug discovery, and the efficiency of BOF significantly degrades compared to static instrumentation, with a performance of only a quarter of the latter. This makes it challenging to assess anti-fuzzing techniques against BOF in these benchmarks, especially when the number of bugs originally found by BOF is limited. Due to these restrictions, we chose to use the LAVA-M benchmark, where bugs directly trigger segmentation faults and are easily discovered by BOF.

Even though the LAVA-M benchmark contains thousands of bugs, BOF techniques can only uncover a few unique bugs per buggy binary. Counting the number of bugs found becomes insignificant, despite being considered the ground truth for fuzzer evaluation. Instead, we measure the time it takes for fuzzers to find the first bug in each buggy binary within 48 hours. This metric better illustrates the bug-finding capabilities of BOF in the LAVA-M benchmark.

Results: Table 1 presents the average time taken by five fuzzers to find a single bug in the LAVA-M benchmark. From the table, we observe that all fuzzers can find at least one bug in the unprotected programs within 48 hours. Notably, QSYM finds the bug in just a few minutes, much faster than other fuzzers. This can be attributed to the design of the LAVA-M benchmark. The bugs in LAVA-M are based on integer comparisons, and if an input bypasses the comparison, the corresponding bug is triggered. This mechanism benefits fuzzers capable of solving constraints, explaining why QSYM, as a

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	AH	all
base64						
AFL	12h54m	√	√	√	17h57m	√
AFLFast	13h8m	√	√	√	13h20m	√
Hfz-QEMU	1h22m	√	√	√	1h13m	√
Hfz-PT	5h23m	√	√	√	3h51m	√
QSYM	2m	√	√	√	7h19m	√
md5sum						
AFL	36h32m	√	√	√	√	√
AFLFast	9h20m	√	√	√	11h36m	√
Hfz-QEMU	-	-	-	-	-	
Hfz-PT	-	-	-	-	-	
QSYM	51m	√	√	√	12h15m	√
who						
AFL	3h8m	√	√	√	7h25m	√
AFLFast	6h37m	√	√	√	8h53m	√
Hfz-QEMU	25m	2h9m	√	6h32m	2h24m	√
Hfz-PT	4h31m	11h45m	√	√	4h41m	√
QSYM	1m	√	√	√	6h30m	√
uniq						
AFL	7h19m	√	√	√	9h29m	√
AFLFast	6h47m	23h59m	√	√	10h42m	√
Hfz-QEMU	4m	10h55m	√	9h3m	6m	√
Hfz-PT	2h48m	16h20m	√	17h56m	2h17m	√
QSYM	5m	√	√	√	22h4m	√

hybrid fuzzer with symbolic execution capabilities, outperforms other mutational fuzzers.

On the other hand, when anti-fuzzing defenses are applied, some fuzzing campaigns exceed the 48-hour time limit without finding any bugs. The remaining campaigns uncover bugs, but it takes significantly more time compared to the corresponding unprotected programs.

Notably, we found that Honggfuzz is incompatible with the md5sum target in LAVA-M, as it mistakenly considers handled errors as crash signals. HonggFuzz generates millions of false positive crash seed files, making it challenging to identify the correct crash seeds from such a large pool. As a result, we had to discard this combination of fuzzer and target.

In general, passive detection techniques and the landing space successfully impede all fuzzing campaigns, as the time taken to find a bug increases after applying these techniques. Similar to **RQ 1**, the anti-hybrid technique only hampers hybrid fuzzing and has no impact on mutational fuzzers. When all anti-fuzzing techniques in No-Fuzz are applied, none of the fuzzing campaigns can find a bug. The evaluation confirms the effectiveness of No-Fuzz in preventing different fuzzers from discovering bugs in protected programs.

C. PERFORMANCE & STORAGE OVERHEAD OF NO-FUZZ

We were inspired by the observation that the size of input files can impact the performance overhead accordingly. Generally, larger inputs invoke more functions and take longer to process. To ensure fairness, we prepared two sets of input files. One set contains small invalid files that quickly trigger errors in the programs, while the other set consists of valid samples of different sizes to trigger normal functionalities. The evaluation results consider the average execution time with both sets of input samples. Additionally, we considered that the overhead would be less significant for large and complex programs. Therefore, we categorized the programs into two groups based on their size and average execution time to mitigate bias in the evaluations. The evaluation results of each group will be analyzed separately.

Performance overhead: According to Table 2, passive detection techniques result in approximately 10-20% performance overhead for small binaries and less than 1% for large binaries. Despite the relatively large overhead for small programs, the absolute latency is only around 5 ms, which regular users typically won't notice. Timing gap detection introduces slightly more latency than the daemon process, likely due to false positives. Similarly, the landing space and anti-hybrid techniques introduce overhead proportional to the size and complexity of the programs. Small programs experience a 40.9-55.3% increase in latency, while for large programs, the proportion decreases to around 2%. The overall overhead is 130.0% for small programs and 3.7% for large programs.

Storage overhead: From Table 2, passive detection techniques consume storage ranging from about 1 KB to 50 KB (10 KB on average), but they are all less than 1% of the original size of the protected programs. The landing space inserts fake blocks based on the number of functions in the protected binary. Consequently, the overhead decreases with fewer functions in the original program, ranging from 0.3 MB to 1 MB (0.8 MB on average) for different programs. Anti-hybrid techniques contribute to storage overhead from 0.04 MB to 0.86 MB, which accounts for only 2.1% of the small program sets and a negligible 0.8% for large programs. The overall storage overhead is around 1.1 MB for small programs and 3.1 MB for large programs. Despite the relatively high overall performance and storage overhead for small programs, it is unnecessary to enable all anti-fuzzing techniques in real-world scenarios. For consideration of overhead, we suggest small programs only adopt the lightweight passive detection methods.

Comparisons with prior works: To demonstrate the value of the landing space, we evaluate existing anti-coverage techniques from previous works. The default configurations of ANTIFUZZ and FUZZIFICATION involve a fixed number of fake blocks, resulting in stable storage overhead of 20 MB and 1.2 MB, respectively. The storage advantage of No-Fuzz is more pronounced for small binaries, as they have fewer fake blocks. It is worth noting that the storage overhead of FUZZIFICATION is significantly smaller than that of ANTIFUZZ. However, our experiments reveal that the default configuration of FUZZIFICATION is insufficient to saturate the fuzzers' bitmaps. Thus, the actual storage overhead of the effective configuration of FUZZIFICATION should be even larger than the current 1.2 MB.

D. ANTI-FUZZING EFFICACY

To assess the effectiveness of different anti-fuzzing techniques, we introduce a new metric called "anti-fuzzing efficacy." This metric quantifies the relationship between the anti-fuzzing effects (coverage reduction) and the associated storage and performance overhead. As anti-fuzzing

TABLE 2. Overhead(CPU) of No-Fuzz and Anti-Coverage Techniques of ANTIFUZZ and FUZZIFICATION on Real-World Programs

	TG	DP	LS	AH	All	AF(cov)	FZ(cov)	Reference
CPU								
Small	6.8ms(21.4%)	3.4ms(10.6%)	13.1ms(40.9%)	17.7ms(55.3%)	41.6ms(130.0%)	11.3ms(35.3%)	14.7ms(45.9%)	32.0ms
Large	23.5ms(1.1%)	10.7ms(0.5%)	43.8ms(2.0%)	53.5ms(2.5%)	81.4ms(3.7%)	15.6ms(0.7%)	44.6ms(2.1%)	2156.2ms
Storage								
Small	8.4K(0.2%)	10.5K(0.3%)	0.8M(25.9%)	65.7K(2.1%)	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%)	0.9M(0.8%)	3.1M(2.9%)	22.3 M (21.0%)	1.27 M (1.2%)	106.5M

TABLE 3. Space and Performance Efficacy of Different Anti-Fuzzing Techniques Against Four Fuzzers

	TG	DP	LS	AH	All	AF(cov)	FZ(cov)
Performance (#branches / ms)							
AFL	183.9	559.3	84.2	3.9	28.2	73.1	11.5
AFLFast	165.0	509.3	61.9	7.3	53.9	66.5	5.3
Hfz-Q	158.5	530.1	86.2	11.3	83.5	76.8	17.0
Hfz-P	134.5	428.4	62.6	14.1	107.4	61.8	5.0
QSYM	226.8	684.9	120.7	37.1	146.8	102.1	14.8
Avg	173.7	542.4	83.1	14.9	84.0	133.2	10.7
Storage (#branches / kB)							
AFL	148.8	181.1	1.4	0.3	0.8	0.07	0.14
AFLFast	133.6	164.9	1.0	0.5	1.6	0.06	0.15
Hfz-Q	128.3	171.6	1.4	0.8	2.4	0.07	0.20
Hfz-P	108.9	138.7	1.0	1.0	3.1	0.05	0.06
QSYM	183.6	221.8	2.0	2.8	4.3	0.09	0.17
Avg	140.6	175.6	1.4	1.1	2.5	0.07	0.14

techniques enhance the defensive capability by inserting additional code into protected programs, a larger amount of defensive code promises more effective defense. An ideal anti-fuzzing technique should introduce minimal overhead while maintaining a high level of defensive capability. Therefore, evaluating the defense capability per unit storage/performance cost provides a better measure of the anti-fuzzing effects. Specifically, we calculate the efficacy by dividing the number of coverage reductions by the amount of defensive code per kilobyte and the reduction in latency per millisecond. As a reference, we also calculate the efficacy of ANTIFUZZ and FUZZIFICATION .

Table 3 demonstrates that passive detection techniques exhibit the highest performance and storage anti-fuzzing efficacy. These techniques are the most cost-effective in countering BOF. Moreover, since fixed defensive code is inserted into the protected programs, the efficacy remains relatively stable across all evaluated programs, with similar orders of magnitude.

The performance efficacy of the landing space is approximately 15–48% compared to passive detection techniques, while the storage efficacy is less than 1% of the passive detection methods. Despite the lower efficiency of the landing space, it can serve as a complementary technique to passive detection. Similarly, the efficacy of Anti-hybrid techniques is only significant for hybrid fuzzers and is less effective compared to other techniques. However, as discussed in Section V, relying on a single defensive technique can be weak against adversaries, making it worthwhile to deploy different techniques if the overhead is manageable.

For reference, we evaluate the anti-coverage techniques of ANTIFUZZ and FUZZIFICATION . The performance efficacy of the landing space is similar to that of ANTIFUZZ, while FUZZIFICATION is less efficient due to the insufficient number of blocks. Regarding storage efficacy, the landing space outperforms these anti-coverage techniques by 10–20 times, resulting in an average reduction of approximately 92.2% in storage cost. This demonstrates that No-Fuzz is a more practical approach, effectively utilizing storage while providing adequate anti-fuzzing protection.

V. DISCUSSION

While our design has demonstrated effective and efficient performance, there is room for further improvement. We view our design as a complement to prior works and recognize the value of certain designs in those works that should be considered for future development of anti-fuzzing tools, such as the installation of a signal handler to conceal crashes.

These concerns will be elaborated on in our discussion. In the subsequent sections, we will delve into the robustness of anti-fuzzing techniques and highlight the advantages they offer over obfuscation. This is important due to the potential inherent issues (i.e., robustness) and the possibility that obfuscation techniques may serve as substitutes in certain scenarios.

Robustness of anti-fuzzing techniques: A key concern regarding anti-fuzzing is its robustness in the face of determined attackers. When the details of the defense mechanism are known, skilled attackers can conduct manual analysis to circumvent it. Prior works have suggested using obfuscation techniques as a countermeasure against reverse engineering. However, even obfuscation techniques can be thwarted by experienced attackers. There is a debate regarding the necessity of making anti-fuzzing techniques robust against manual analysis, it may not be essential to specifically address this aspect of robustness.

Anti-fuzzing techniques aim to introduce additional obstacles (such as time, resources, and knowledge) that adversaries must overcome when attempting to fuzz a protected program. These techniques are particularly effective in defending against large-scale untargeted fuzzing tasks, which do not warrant the manual analysis of individual binaries for attackers. Furthermore, anti-fuzzing techniques raise the bar for successful BOF by necessitating an understanding of both anti-fuzzing and reverse-engineering techniques. The defensive measures can also decrease the likelihood of protected binaries being selected as targets for BOF. Given the aforementioned reasons, we assert that the ultimate purpose of anti-fuzzing techniques is not undermined by reverse engineering.

Anti-fuzzing or obfuscation: Obfuscation has been considered as a potential solution to anti-fuzzing, as it is a well-developed technique with strong community support. However, prior works have conducted experiments that question the effectiveness of obfuscation in anti-fuzzing [17], [23]. We aim to revisit their arguments and provide additional experiments to demonstrate that obfuscation techniques can be effective in certain cases.

Interestingly, there are already obfuscation techniques designed specifically to counter symbolic executions, similar to anti-hybrid techniques [4], [39]. Furthermore, obfuscation techniques that involve self-modifying code can be particularly powerful against BOF. Self-modifying code is commonly used in packing and encryption, allowing the reuse of memory space by overwriting existing opcodes with new instructions.

However, a challenge arises when multiple functions are overwritten within a self-modifying block, as they share the same memory address. Most fuzzers rely on hashing function block addresses, causing the overwritten functions to be identified as a single function. As a result, the coverage information of the overwritten functions is lost.

We conducted experiments using BOF on dummy programs protected by self-modifying code. The results demonstrate that BOF cannot be successfully performed on programs with self-modifying code. Additionally, a study by Raffetseder et al. [30] highlights that self-modifying code significantly slows down translations in emulators. This further confirms the anti-fuzzing effectiveness of obfuscation against BOF, as observed with tools like `afl-qemu` and `honggfuzz-qemu`.

Fortunately, self-modifying code is not commonly employed by developers. Commercial software rarely utilizes self-modifying code due to the risk of false positives as malicious attempts and the introduction of new bugs through risky modifications. In summary, anti-fuzzing techniques address the limitations of static obfuscation techniques against fuzzers. We argue that future anti-fuzzing research should focus on static techniques without relying on self-modifying code.

VI. CONCLUSION

This paper presents the design and implementation of No-Fuzz, a prototype tool that integrates practical and fully automated anti-fuzzing techniques. We optimize storage by inserting fake blocks at a granular level, reducing unrealistic requirements. We enhance computational efficiency by using a lightweight wrapping function and input-tainted constants, reducing manual efforts compared to previous methods. Additionally, we introduce passive detection methods to identify execution environments and apply mitigation techniques for binary-only fuzzing.

Our evaluations demonstrate that No-Fuzz significantly reduces fuzzers' branch coverage and hinders bug discovery in the LAVA-M dataset. We propose the "anti-fuzzing efficacy" metric, showing that No-Fuzz provides equal or higher protection with lower overhead compared to previous approaches. Notably, our solution achieves an impressive average reduction in storage cost of 92.2% compared to prior works. In summary, we emphasize the importance of minimizing overhead and embracing automation in the anti-fuzzing domain. By doing so, we pave the way for more practical and effective techniques and aspire to inspire further progress in this field.

REFERENCES

[1] "Google fuzzer test suite," Accessed: Mar. 12, 2022. [Online]. Available: https://github.com/google/fuzzer-test-suite

[2] "A library for coverage-guided fuzz testing," Accessed: Oct. 23, 2020. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[3] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, "Efficient detection of split personalities in malware," in *Proc. Netw. Distrib. Syst. Secur.*, 2010, pp. 1–16.

[4] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proc. Annu. Conf. Comput. Secur. Appl.*, 2016, pp. 189–200.

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2016, pp. 1032–1043.

[6] M. Böhme et al., "Directed greybox fuzzing," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2017, pp. 2329–2344.

[7] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1621–1633.

[8] H. Chen et al., "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2018, pp. 2095–2108.

[9] Y. Chen et al., "SAVIOR: Towards bug-driven hybrid testing," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1580–1596.

[10] A. Dewitz and W. Olofsson, "The hare, the tortoise and the fox: Extending anti-fuzzing," M.S. thesis, Blekinge Inst. Technol., 2022.

[11] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1497–1511.

[12] B. Dolan-Gavitt et al., "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 110–121.

[13] E. Edholm and D. Göransson, "Escaping the fuzz-evaluating fuzzing techniques and fooling them with anti-fuzzing," 2016.

[14] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *Proc. IEEE Secure Develop. Conf.*, 2020, pp. 23–30.

[15] Google, "A scalable fuzzing infrastructure," Accessed: Oct. 23, 2020. [Online]. Available: https://github.com/google/clusterfuzz

[16] Google, "Syzkaller found bugs - Linux kernel," Accessed: Oct. 23, 2020. [Online]. Available: https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md

[17] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "Antifuzz: Impeding fuzzing audits of binary executables," in *Proc. USENIX Secur.*, 2019, pp. 1931–1947.

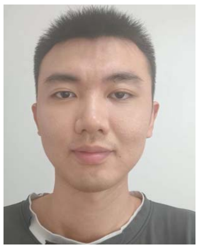
[18] M. Hafiz and M. Fang, "Game of detections: How are security vulnerabilities discovered in the wild?," *Empirical Softw. Eng.*, vol. 21, pp. 1920–1959, 2015.

[19] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," in *Proc. ACM Meas. Anal. Comput. Syst.*, 2020, pp. 1–29.

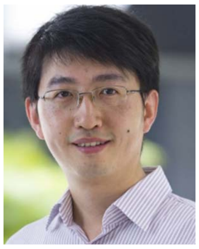
[20] Z. Hu, Y. Hu, and B. Dolan-Gavitt, "Chaff bugs: Deterring attackers by making software buggier," 2018, *arXiv:1808.00659*.

[21] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1613–1627.

[22] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: Directed grey-box fuzzing with provable path pruning," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 36–50.
[23] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *Proc. USENIX Secur.*, 2019, pp. 1913–1930.
[24] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating emulation-resistant malware," in *Proc. ACM Workshop Virtual Mach. Secur.*, 2009, pp. 11–22.
[25] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proc. ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 475–485.
[26] Y. Li et al., "Vall-nut: Principled anti-grey box - fuzzing," in *Proc. IEEE Int. Symp. Softw. Rel. Eng.*, 2021, pp. 288–299.
[27] Y. Li et al., "UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *Proc. USENIX Secur.*, 2021, pp. 2777–2794.
[28] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2011, pp. 338–357.
[29] P. Barton, L. M. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, pp. 32–44, 1990.
[30] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *Proc. Int. Conf. Inf. Secur.*, 2007, pp. 1–18.
[31] M. Rash, "A collection of vulnerabilities discovered by the AFL fuzzer," Accessed: Sep. 13, 2020. [Online]. Available: https://github.com/mrash/afl-cve
[32] S. Rawat, V. Jain, A. J. S. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. Netw. Distrib. Syst. Secur.*, 2017, pp. 1–14.
[33] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. USENIX Secur.*, 2017, pp. 167–182.
[34] N. Stephens et al., "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. Netw. Distrib. Syst. Secur.*, 2016, pp. 1–16.
[35] R. Swiecki, "Honggfuzz," 2020. Accessed: Aug. 14, 2023. [Online]. Available: https://github.com/google/honggfuzz
[36] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek, "Hackers vs. testers: A comparison of software vulnerability discovery processes," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 374–391.
[37] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 497–512.
[38] Y. Wang et al., "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *Proc. Netw. Distrib. Syst. Secur.*, 2020, pp. 1–17.
[39] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *Proc. 16th Eur. Symp. Res. Comput. Secur.*, 2011, pp. 210–226.
[40] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. USENIX Secur.*, 2018, pp. 745–761.
[41] M. Zalewski, "American fuzzy lop," 2019. Accessed: Aug. 14, 2023. [Online]. Available: http://lcamtuf.coredump.cx/afl
[42] M. Zalewski, "Technical 'whitepaper' for afl-fuzz," 2019. Accessed: Aug. 14, 2023. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt
[43] 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. IEEE Symp. Secur. Privacy*, 2021, pp. 659–676.
[44] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei, "FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing," in *Proc. USENIX Secur.*, 2022, pp. 3699–3715.
[45] Z. Zhou, C. Wang, and Q. Zhao, "No-fuzz: Efficient anti-fuzzing techniques," in *Proc. Secur. Privacy Commun. Netw.*, 2023, pp. 731–751.
[46] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proc. USENIX Secur.*, 2020, pp. 2255–2269.



ZHENGXIANG ZHOU received the BE degree in computer science and technology from The Chinese University of Hong Kong, Shenzhen, China, in 2019. He is currently working toward the Ph.D. degree with the Department of Computer Science with the City University of Hong Kong, Hong Kong. His research interests include fuzzing and software engineering.



CONG WANG (Fellow, IEEE) is currently a Professor with the Department of Computer Science, City University of Hong Kong, Hong Kong. His research interests include data and network security, blockchain and decentralized applications, and privacy-enhancing technologies. He was a co-recipient of the IEEE INFOCOM Test of Time Paper Award 2020, the Best Paper Award of IEEE ICDCS 2020, ICPADS 2018, and MSN 2015, Best Student Paper Award of IEEE ICDCS 2017, Outstanding Researcher Award in 2019, Outstanding Supervisor Award in 2017, and the President's Awards with the City University of Hong Kong in 2016 and 2019, respectively. He is a Founding Member of the Young Academy of Sciences of Hong Kong and a Research Fellow of the Hong Kong Research Grants Council. He is the Editor-In-Chief of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING .