Robust, Efficient, and Widely Available Greybox Fuzzing for COTS Binaries with System Call Pattern Feedback

Jifan Xiao[1], Peng Jiang[2], Zixi Zhao[1], Ruizhe Huang[1], Junlin Liu[1], and Ding Li[1]

[1] *Key Laboratory of High Confidence Software Technologies, Peking University*
[2] *Southeast University*

Abstract

Currently, greybox fuzzing is a crucial technique for identifying software bugs. However, applying greybox fuzzing to Commercial-Off-the-Shelf (COTS) binaries is still a difficult task because gathering code coverage data is challenging. Existing methods for collecting code coverage in COTS binaries often lead to program crashes, notable performance reductions, and limited compatibility with various hardware platforms. As a result, none of the current approaches can effectively handle all COTS binaries.

This paper introduces a new feedback mechanism called system call pattern coverage, which is designed to support binaries that cannot be handled by existing approaches. Unlike other methods, system call pattern coverage does not involve rewriting binaries, using emulators, or relying on hardware such as Intel-PT. As a result, it enables fuzzing of binaries without the risk of breaking target applications, slow performance, or the need for specific hardware. To demonstrate the effectiveness of this mechanism, we developed fuzzers called SPFuzz and SPFuzz++ and conducted an evaluation using 29 real-world benchmarks. The results of our evaluation show that SPFuzz and SPFuzz++ perform comparably to conventional code coverage guidance and are capable of identifying new bugs even without access to the source code. In fact, we discovered six new CVEs in commercial applications like CUDA using SPFuzz.

1 Introduction

Greybox fuzzing, also known as guided fuzzing, has become a crucial technique for identifying software bugs [65]. AFL, a well-known greybox fuzzer, has successfully detected numerous CVEs in real-world software, such as the Heartbleed bug and other zero-day vulnerabilities [26, 47]. However, effectively applying greybox fuzzing to binary-only commercial off-the-shelf (COTS) software remains a challenging issue due to the difficulty in obtaining code coverage feedback [46]. Unlike open-source software, which can use compilers to instrument probes and gather code coverage information, binary-only programs are hard to instrument, particularly when they are obfuscated [30]. Despite numerous research efforts to collect code coverage information from binary-only programs, they still struggle to cover all common use scenarios [54].

Current approaches to fuzz binary-only COTS programs often result in program crashes, significant slowdowns, and limited applicability to different hardware platforms. Existing binary-only greybox fuzzers can be classified into three categories: static rewriting-based, emulator-based, and Intel-PT-based techniques. However, each of these approaches has its limitations. Static rewriting-based approaches [30, 58] can cause target programs to crash. Emulator-based techniques [4, 22, 40] can slow the target program 6-10 times [53]. Besides, emulators often have difficulties handling the latest hardware features. For example, we notice that the widely used emulator, QEMU [22], cannot execute X86 programs that contain AVX instructions [13]. Intel-PT-based techniques [8] can introduce hardware dependencies, thus limiting their scope of application. Furthermore, Intel-PT may generate a large number of events in a short period of time [14]. Processing too many events can substantially slow down the fuzzing speed and exhaust the memory of the fuzzers [66]. Therefore, we can still observe Intel-PT based fuzzers crash or run slowly on certain apps. In summary, it is necessary to develop new fuzzing feedback mechanisms that can support all COTS binaries on different hardware platforms. Unfortunately, to our knowledge, there are no such feedback mechanisms available in the community.

In this paper, we introduce a novel feedback mechanism called the **system call pattern coverage** to address the limitations in current approaches[1]. Unlike static rewriting-based and emulator-based methods, our feedback mechanism can support all COTS binaries that use system calls. Furthermore, it does not significantly slow down the target programs as emulators do. Our system call pattern coverage is not depen-

[1] In this paper, we use *system call* and *syscall* alternatively.

dent on Intel-PT, which means that it can support all major CPUs and cloud platforms, providing a wider scope of applications. In summary, our system call pattern coverage can enable greybox fuzzing for COTS binaries that cannot be supported by current approaches in different scenarios, making it an efficient alternative when code coverage is not available.

Our main insight is that *people can infer code coverage information from system calls and their corresponding parameters*. Programs require interactions with operating systems through various system calls to achieve their desired functionalities. Therefore, system calls are situated in various positions and can function as covert instrumentation points. By implementing an appropriate encoding and analysis method, concealed execution information can be extracted from system call sequences and reveal the code executed during fuzzing.

Using system call patterns to guide greybox fuzzing effectively is not as straightforward as it may appear. There are two main challenges that arise from this approach. **First, system call sequences are ambiguous.** They do not have a one-to-one correspondence with the code blocks. Different code blocks can generate the same system call trace, and the same code blocks can generate different system call sequences depending on the inputs. To resolve this ambiguity, we would need to use heavy static analysis techniques such as symbolic execution [45]. However, these techniques are too slow for efficient fuzzing. **Second, the effectiveness of the system call parameters varies between different applications.** A system call parameter that is useful for fuzzing one application may cause instability(e.g. get random values) for another application. Therefore, there is no fixed set of parameters that can work well for all target programs.

We propose novel designs for system call pattern coverage to address the two challenges mentioned above. To tackle the first challenge, we introduce a novel N-gram system call pattern that accurately reflects the changes in code paths during fuzzing without depending on heavy static analysis techniques to obtain precise code coverage data. Our key idea is that we only need to know which lines of code are executed during fuzzing rather than the exact code path. Therefore, we can use the N-gram system call patterns to locate the code segments that have been run. Moreover, we can quickly distinguish different system call patterns by computing and comparing their hash code, reducing the runtime overhead for fuzzing the target program. To address the second challenge, we propose an automated technique that eliminates parameters that cause unstable fuzzing outcomes based on the execution of seeds. In our evaluation, this technique ensures effectiveness while also increasing fuzzing speed by up to 45 times.

We implemented fuzzers, SPFuzz (with AFL) and SPFuzz++ (with AFL++), based on our system call pattern coverage to evaluate its effectiveness. We perform a thorough evaluation of SPFuzz and SPFuzz++ on 29 real-world applications, both open-source and closed-source. Our evaluation shows that SPFuzz and SPFuzz++ are capable of fuzzing all applications that use system calls. In comparison, six baseline binary-only fuzzers, AFL-QEMU, AFL++QEMU, ZAFL, AFL++Nyx, StochFuzz, and PTFuzzer, are unable to fuzz four to 11 applications. Furthermore, SPFuzz is up to 41 times faster than AFL-QEMU. Our evaluation also reveals that the system call pattern coverage is as effective as traditional code coverage in guiding the fuzzing process to discover new codes. On average, a fuzzer guided by system call patterns achieves up to 10.3% higher branch coverage than binary-only fuzzers guided by conventional code coverage, for the same base fuzzers. Using SPFuzz, we have discovered six new CVEs in four applications, including one in the *CUDA* toolset. This CVE cannot be detected by AFL-QEMU or ZAFL since they cannot run the newest *CUDA-nvdisasm* efficiently. As for SPFuzz++, it achieves comparable and even better coverage and speed than all baselines, including AFL++QEMU and AFL++Nyx. Overall, our evaluation demonstrates that system call pattern coverage is a valuable alternative to code coverage.

We summarize the contributions of this paper as follows:

- We are the first to propose system call pattern coverage, which provides an effective alternative way to provide feedback for greybox fuzzing when code coverage is not available.
- We implemented the first binary-only fuzzers based on this new coverage, SPFuzz and SPFuzz++, which can fuzz more COTS binaries than existing approaches.
- We systematically evaluated SPFuzz with 18 realistic open-source and 11 closed-source applications.

Open Science. Our code and data are available at `https://github.com/Nova-xiao/SPFuzz` or `https://doi.org/10.5281/zenodo.14614073` and are ready for continuous development by the community.
Ethics. We have reported all the vulnerabilities discovered by SPFuzz to the authors/owners of the vulnerable software.

2 Background

2.1 Greybox Fuzzing

The process of greybox fuzzing or guided fuzzing [46] involves using the running information of a target program to generate input that can discover new code. Algorithm 1 outlines the general process of a guided fuzzer, which is essentially a loop that takes in a target program p and a set of initial inputs called seeds. Before running the loop, the algorithm initializes the queue Q with the seeds (line 1). During each iteration of the loop, the fuzzer fetches an input from Q, runs it with the target program p, and collects the execution time information (line 4). Based on this information, the fuzzer

assigns a priority score called energy to the input (line 5) and mutates the current input to generate a new input (line 6). The higher the energy score, the more new inputs will be generated from the current input. Finally, the new inputs are added to Q, and the fuzzing loop repeats.

In modern guided fuzzers, the runtime information used is code coverage [46]. This can be measured in various ways, such as statement coverage [65] or branch coverage [50]. The code coverage here is essentially a set of addresses or line numbers that show which instructions have been executed during runtime. Equation 1 summarizes this, where l_i represents the address or line number of an instruction that has been executed.

$$CC = \{l_0, l_1, ..., l_n\} \tag{1}$$

Obtaining code coverage information can be done by instrumenting probes to the target program. However, it is particularly challenging to do this with binary-only COTS programs. As a solution, *our paper aims to propose a new method for runtime information feedback that can be used instead of code coverage when instrumentation is not an option.*

Algorithm 1: The Main Loop of Guided Fuzzing

```
   Input  : the target program p and a set of seeds S
1  Q ← S
2  while Q ≠ ∅ do
3  |   q ← Q.pop()
4  |   info ← p(q)
5  |   e ← Energy(info)
6  |   q' ← Mutate(q, e)
7  |   if e > 0 then
8  |   |   Q.append(q')
9  |   end
10 end
```

2.2 Binary-Only Fuzzing

One of the main challenges when it comes to binary-only guided fuzzing is the necessary code coverage information. To achieve this, there are currently three main techniques that can be utilized: emulation [4, 22], static binary rewriting [54, 58, 67], and Intel-PT-assisted tracing [8, 25]. However, each of these methods comes with its own set of drawbacks when it comes to realistic fuzzing tasks. In the following paragraphs, we will briefly explore the limitations of each of these approaches.

Emulation The key limitation of emulation-based approaches is that they significantly slow down the target program by 6x on average and up to 100x [53]. Note that this is a fundamental limitation for emulators, since they use software logic to emulate hardware behaviors, which will inevitably slow down the execution speed of programs. Furthermore, building emulators for various hardware architectures is also challenging. It is difficult to ensure the soundness of emulators in practice. For example, QEMU does not support x86 AVX instructions [12]. Implementing QEMU that supports AVX instructions and migrating fuzzers, such as AFL-QEMU [4], is labor intensive. In our experiments, many of our open-source benchmarks will result in unexpected crashes in QEMU under the default compilation configurations.

Static Rewriting. The main issue with static rewriting techniques is their tendency to disrupt the logical flow of the targeted programs [30]. This is mainly due to the absence of an effective method for modeling the addresses of indirect jumps. It is essential to accurately track the target of such jumps in order to statically instrument binaries. However, existing static analysis techniques cannot achieve this objective. Consequently, current static rewriting techniques frequently result in the alteration of target binaries. For example, in our evaluation, the state-of-the-art instrumentation-based approach, ZAFL [54], still caused crashes in 28% of the binaries compiled from 18 open-source applications. We have reported to ZAFL developers and confirmed that their base rewriter Zipr [36] may run out of memory during analysis or fail to resolve stack pointers on large binaries. It is also confirmed that Zipr may hit internal assertions and fail, or alter the semantics of programs with reasonable sizes, such as changing the dynamic exception specifications. Although RetroWrite [30] claims to be able to handle the indirect jumps, it requires the symbol table information and is therefore limited to non-stripped binaries.

Intel-PT Assisted Tracing. A limitation of Intel-PT assisted tracing is that it only works for binaries run on Intel CPUs. However, many other CPU architectures, such as ARM and RISC-V, are widely used today. For example, ARM [61] dominates the smartphone market and has 15% of the PC market share, according to a recent report [11]. Unfortunately, none of the commercial ARM processors have implemented a mechanism equivalent to Intel-PT. Similarly, RISC-V [63], another emerging CPU architecture, does not provide such a mechanism in its commercial implementations. Therefore, Intel-PT assisted tracing cannot support the large amount of COTS software for these non-Intel CPU architectures.

Furthermore, the use of Intel-PT can result in a significant number of tracing events being generated within a short time frame, causing inefficiencies in Intel-PT assisted fuzzers. In practical scenarios, it is not uncommon for Intel-PT to generate trace data of several hundred megabytes per CPU core per second [14]. Processing such a large volume of tracing events within a limited period of time consumes a significant amount of system resources, ultimately slowing down the fuzzing process. Our experiments reveal that PTFuzzer, one of the most commonly employed Intel-PT assisted fuzzers, can experience a slowdown in fuzzing speed ranging from 20% to 84 times when compared to alternative approaches. Additionally, the high number of tracing events can cause an

overflow of the event buffer in the fuzzer, resulting in crashes. For example, our experiments indicate that PTFuzzer crashes for 40% of the applications due to event buffer overflow, even after enlarging the event buffer to 4GB.

3 A Motivating Example

The key insight of this paper is that people can infer the code coverage by analyzing system calls and their corresponding parameters. To utilize hardware and system resources, modern user-level applications must execute system calls and provide parameters. Patterns of these system calls and their parameters can indicate how the application behaves while running. By identifying patterns in the system call logs, we can identify which code paths have been taken by an execution.

In this section, we will illustrate the insight and challenges of our approach using a real-world example. Figure 1 displays a simplified example from Xpdf, a well-known PDF viewing toolkit, and one of the applications in our benchmark for evaluation. The code snippet ❶ represents the main loop of the Xpdf-pdftotext tool. Initially, it verifies the settings and decides whether to print the debugging information or not (Lines 10-14). Then, it uses the `displayPage` function to manage each page (Lines 15-17). This function will read or write the PDF content according to the content formats. After this, the `doneWithPage` function will be invoked to perform the post-processing task, such as closing the PDF file if the end has been reached (Line 20).

The snippet ❷ in Figure 1 displays the potential system calls that can be produced by each line of code in ❶. The *Lines* column refers to the line number in ❶, while the column *Possible System Calls* indicates the corresponding system call trace that can be generated. The symbol − represents the connection between sequential system calls, whereas the symbol ? in the figure implies that the corresponding system call may or may not appear based on different inputs. For example, Line 15 can generate two possible system call sequences: *lseek* → *write* and *lseek* → *read* → *write*.

In the figure, snippets ❸ and ❹ display the system call sequences of three different executions, along with their corresponding code paths in ❶. By analyzing these traces, we can determine which code path is taken by different inputs. In Syscall-trace 1, the *fstat* system call corresponds to Line 10 in ❶, while the three system calls *lseek, read (ret = 0)*, and *write* correspond to Line 15. From this information, we can infer that the main loop is executed once and successfully displays the image at Line 15, since the read system call returns 0.

Similarly, we can infer from Syscall-trace 2 that the main loop was executed twice. This is because the *fstat* function corresponds to Line 10, while the sequence of *lseek* → *write* corresponds to Line 15. Note that *lseek* → *write* can respond to Lines 11, 13, and 15. However, since Lines 11, 12, and 13 are in the same basic block, they must be executed together. This means that if Line 11 or Line 13 is executed, the system call trace should be *lseek* → *write* → *write* → *lseek* → *write*. In our case, there is an *fstat* function, which corresponds to Line 10, between the two sequences of *lseek* → *write*. Therefore, the *lseek, write* sequence can only be generated by Line 15 in two separate iterations.

In Syscall-trace 3, the parameters of system calls help distinguish execution results. An example of this is shown in Line 18, which is a branch statement that depends on the return status of Line 15. However, since Line 19 does not generate any system calls, it is impossible for the system call sequences to determine whether the path condition on Line 18 is true or false. To infer the path condition at Line 18, we can rely on the return values and parameters of system calls.

It is important to note that the path condition "$s == 0$" can only be true if *displayPage* returns zero at Line 15. The return value of *displayPage* is inferred from the return value of the *read* system call. When *read* returns -1, *displayPage* cannot return zero because it fails to open the target file. Therefore, in Syscall trace 3, since the return value of *read* is -1, we conclude that the value of *s* cannot be zero, and Line 19 cannot be executed.

After analyzing the case studies mentioned above, we have concluded that the system call patterns have the potential to estimate the code coverage effectively. This discovery inspired us to create system call pattern coverage as a potential alternative to traditional code coverage.

Challenges: When using system call patterns in greybox fuzzing, the main difficulty lies in dealing with the uncertainty surrounding system call sequences. This uncertainty can be divided into two parts: uncertainty in system call sequences and uncertainty in the parameters and return values of system calls. When it comes to system call sequences, multiple lines of code can generate the same system call sequences, while one line of code can generate different system call sequences. For example, as seen in Figure 1, Lines 11, 13, and 15 all generate the same system call sequence. On the contrary, one line of code can generate different system call sequences, as seen with Lines 15. To accurately determine code paths from system call sequences, heavy static analysis techniques such as symbolic execution [38] must be used. However, such techniques are not practical for fuzzing due to scalability issues. Symbolic execution, for example, can take several minutes to analyze a program [38], which is not efficient enough for fuzzing as thousands of seeds need to be run to find bugs.

When it comes to the parameters and return values of the system calls, some of them are helpful for identifying different code paths, while others can make the fuzzing process unstable. For example, the return value of the *read* system call is crucial in distinguishing between syscall-trace 1 and syscall-trace 3 in ❸ of Figure 1. On the other hand, the return value of *mmap* can be random, generating different values for the same input. Thus, we need to carefully select which param-

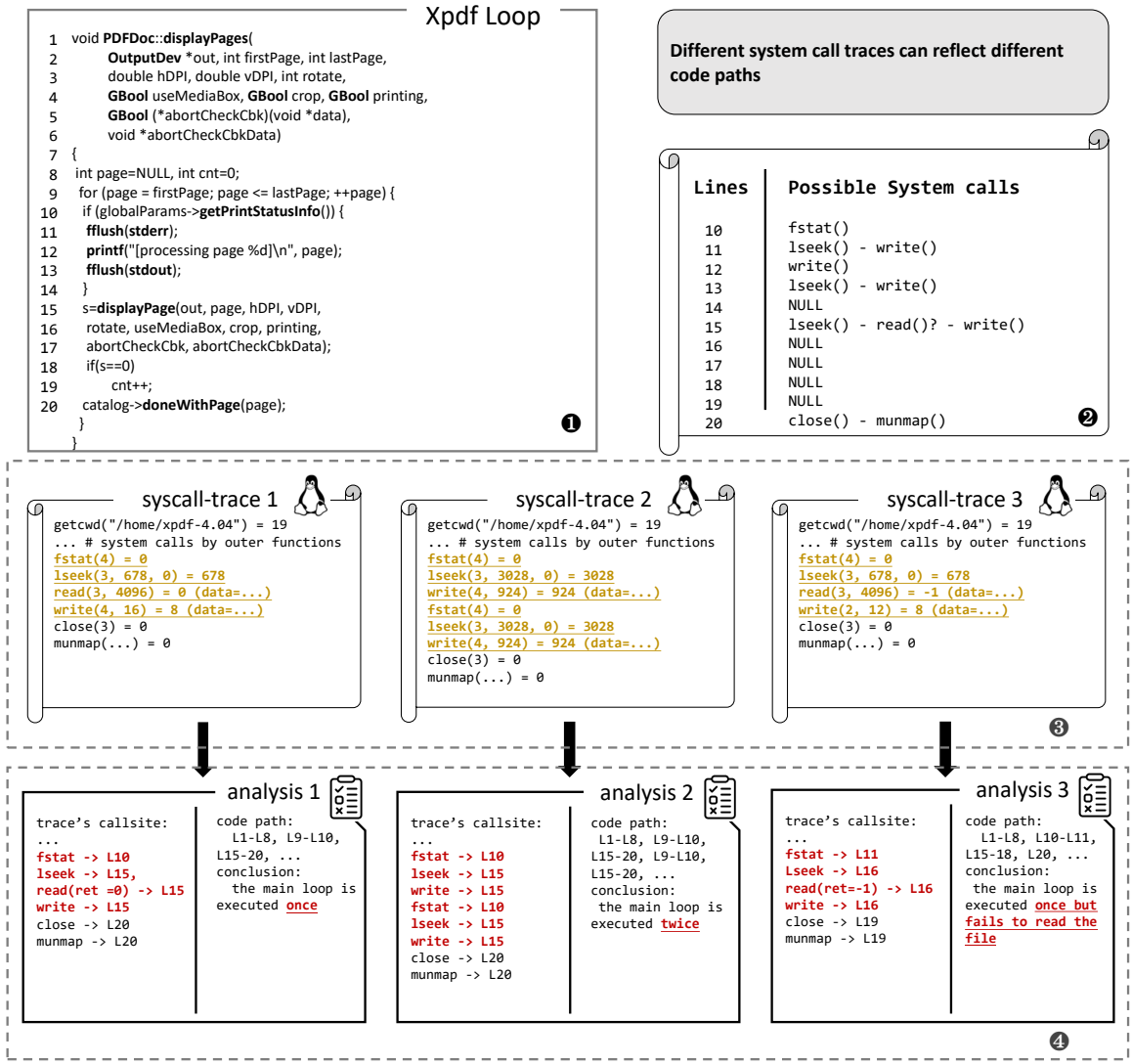


Figure 1: A motivating example that shows how OS-level traces help identify different paths.

eters and return values should be included while generating system call patterns.

Additionally, the usefulness of a parameter or return value can vary between different applications. For example, the *buf* parameter of the system call *getcwd* contains the absolute path name of the current working directory, which is useful for `pdftotext` of Xpdf, since it tells which dynamic libraries are selected to handle different PDF elements. However, including the same parameter in `nginx` can reduce the stability of the fuzzing, as it only contains a randomly generated folder name. Therefore, we cannot rely on a static rule set to determine the usefulness of parameters for system calls, as they can differ depending on the application.

Solutions: Our insight into addressing the first challenge is that we can still effectively lead greybox fuzzing to discover new code without accurately recovering which code path has been executed. In guided fuzzers, code coverage is used to indicate how many lines of new code are discovered by an input. Thus, as long as we find an *easy-to-compute* feedback that effectively approximates how many new codes an input can discover, we may use it to guide greybox fuzzing.

Our insight in addressing the second challenge is that, for a given target program, the usefulness of the parameters of a system call is determined by the semantics of the code. Thus, we can run a set of initial inputs (seeds) to get basic information about the usefulness of the system call parameters. Then, we can build an automated learning method to choose useful parameters based on the information collected from the seeds.

4 System Call Pattern Coverage

In this section, we present the concept of system call pattern coverage, which is an effective feedback for greybox fuzzing in situations where code coverage cannot be obtained. In order to make greybox fuzzing effective, the feedback mechanism needs to meet two requirements: (1) it should be easy-to-compute so that the fuzzer does not become too slow; and (2) it should give enough information about the code that is executed by different inputs. To meet these goals, we propose a novel N-gram system call pattern coverage.

Intuitively, we define our system call pattern coverage as the set of various patterns of N-gram system call sequences that occur during the execution of the target program. Formally, given a system call trace, $S = \langle s_0, s_1, ..., s_l \rangle$, where s_i is a system call event with its parameters and $(l+1)$ is the trace length, we define the N-gram system call pattern coverage as Equation 2, in which h is a hash function, and N is the length of the sliding window used in N-gram. Then we define a system call pattern covered by the execution as a hash value $h(s_i, ...s_{i+N-1})$in Equation 2.

$$SPC_N = \{h(s_0, ..., s_{N-1}), h(s_1, ..., s_N), ...h(s_{l-(N-1)}, ..., s_l)\} \tag{2}$$

In order to determine the coverage of system call patterns, we begin by collecting the system call sequences while running the specific program under analysis (as demonstrated in Section 5.1). Next, we slide a window of size N-gram along the trace (as shown in Section 4.1), computing the hash value of the system calls and their corresponding parameters at each step. Finally, we compute all the hash values obtained during this procedure and regard them as the covered system call patterns during execution.

SPC can be used similarly to the code coverage as demonstrated in Equation 1. If the code coverage is not available, *SPC* can be used instead to give feedback on the fuzzing. For example, the number of newly discovered hash values of N-gram system call patterns ($h(s_i, ..., s_{i+N-1})$) can be used in place of addresses to prioritize inputs.

4.1 Long-short System Call Pattern Coverage

When using the system call pattern coverage defined in Equation 2, *selecting the appropriate N* for the N-gram sliding window is a key challenge. Unlike the line number or address used in code coverage, which is naturally unique for different lines of code and ordered, our system call pattern coverage can have conflicts. Note that the inner elements of SPC_N are not ordered, there may be instances where two different system call sequences generate the same system call patterns, (e.g. *read,write,read,write* and *write,read,write,read* with $N = 2$). In such cases, when it comes to the number of system call patterns, there are pros and cons in having a small or large value for N. If N is too small, there may be more conflicts, since the number of possible system call patterns is limited. However, if N is too large, it may not effectively measure the efficacy of input. For example, if N is set to the length of the entire system call trace, the final SPC_N set in Equation 2 will only have one element. This means that the system call pattern coverage can only determine if a new input can uncover new codes, but not evaluate if one input can find more new codes than another. Furthermore, a large N also means a large state space for *SPC*, which may lead to the state explosion problem, which can reduce the effectiveness of fuzzing.

To address this challenge, we have adopted a long-short combined system call pattern coverage approach. Rather than relying on a single N value, we use a combination of system call patterns discovered by three different lengths of sliding windows. The first length (and the shortest length) is indicated by θ, which is the most stable feature that aims to capture *basic functional units* in a program. We define a basic functional unit as the smallest code block that performs a single task in the program. It could be a function call, a library call, or a series of basic code blocks. Note that in practice, the length of a basic functional unit varies. Therefore, the length of system calls generated by different basic functional units is also different. θ only aims to capture the best length that can reflect the basic functional units in a program. Therefore, θ is different between applications. We will have an automated method to find the best θ for each program, which we will discuss later in this section.

The second length is 2θ, and it is used to evaluate the variations in the context of each basic functional unit. Some basic functional units, such as a function, might be used in various contexts. The 2θ window can help recognize whether a *basic functional unit* is invoked in different contexts.

The third length is the full length of the system trace. This is the bottom-line coverage that captures whether an input can discover new code. Thus, in summary, the long-short system call pattern coverage is defined in Equation 3, where SPC_θ, $SPC_{2\theta}$, and SPC_{full} represent the system call pattern coverage generated with window length θ, 2θ, and the full length of the system call trace, respectively:

$$SPC = SPC_\theta \bigcup SPC_{2\theta} \bigcup SPC_{full}. \tag{3}$$

The value of θ is generated by a dynamic search algorithm. The input to the algorithm, $\mathcal{C}$, is a set of inputs that should cover different usage scenarios of the targets. Formally, $\mathcal{C} = \{seed_1, seed_2, ...seed_n\}$, where $seed_k$ is the kth fuzzing seed file and n is the size of $\mathcal{C}$. This algorithm consists of three steps. First, we construct sets C_1 and C_2 with the form of $\{\{seed_i, seed_j\}...\}$, where $seed_i, seed_j$ are from $\mathcal{C}$. We define $C_1 = \{\{seed_i, seed_j\} | 1 \le i, j \le n, i \ne j\}$, and $C_2 = \{\{seed_m, seed_m\} | 1 \le m \le n\}$. Second, we collect traces by running the target program with the elements from C_1 and C_2 and get system call sequences. Third, we set $\theta = 2$ and double it in every subsequent step until two conditions are

met at the same time. The first condition is that for any element in C_1, the generated $SPC_\theta \bigcup SPC_{2\theta}$ are different. This condition indicates that $SPC_\theta \bigcup SPC_{2\theta}$ should be distinct for different functionalities. The second condition is that for any element in C_2, the generated $SPC_\theta \bigcup SPC_{2\theta}$ with two runs are identical. This condition indicates that $SPC_\theta \bigcup SPC_{2\theta}$ should remain the same for identical inputs.

4.2 System Call Parameter Selection

When using our system call pattern coverage, another challenge is selecting the appropriate set of parameters for each system call. As explained in Section 3, the usefulness of a parameter or return value can differ depending on the application being used. Therefore, a dynamic algorithm is required to choose the set of system call parameters when generating the system call patterns.

At a high level, our method is a learning-based algorithm that gradually removes the parameters that cause instability during the fuzzing process. We use the set of option-inputs mentioned in Section 4.1 as input, and the output is a list of reserved parameters.

The high-level idea of our algorithm is that unstable parameters produce different patterns for identical inputs. If every syscall includes unstable parameters, the number of patterns from two runs of C_1 will be twice the size of C_1. Conversely, if the parameters are stable, the number of patterns from two runs will match the size of C_1. Therefore, the goal of this algorithm is to prune parameters to ensure the number of patterns matches the size of C_1.

Our algorithm consists of three steps. In the first step, we collect system call sequences by running the target program with C_1 twice and get the system call sequences with all parameters. In the second step, we analyze the traces from C_1 by generating system call patterns. If the number of distinct patterns is larger than the size of C_1, it means that the fuzzing process is unstable and we move to the third step. The third step is a *pruning* process, and we prune the least stable parameter to make the fuzzing process stable. We repeat the above steps until the deviation of the number of newly generated system call patterns and the size of C_1 is less than 5% to ensure that the parameters are sensitive enough to find new paths.

When using our parameter selection algorithm, we need to address the problem of determining which parameter is least stable and should be removed during the *pruning* process (the third step). To address this problem, we have developed a heuristic that evaluates the stability of the parameters.

Our heuristic is based on two key insights. First, complex parameters are more likely to be unstable compared to simpler ones. For instance, an array parameter is more prone to instability compared to a boolean parameter. Second, the parameters of a complex system call are also more likely to be unstable. Therefore, we rank the parameters according to their complexity and remove the parameter with the highest complexity during the *pruning* process.

Specifically, the complexity of a parameter consists of two parts. The first-order complexity is defined as the size of the parameter. For example, an 8-bit integer has a complexity of eight. A pointer has the highest first-order complexity. The second-order complexity is defined as the complexity of the corresponding system call of the parameter, and we define the complexity of a system call as its number of parameters. To rank two parameters, we first compare their first-order complexity. If it is the same, we compare the second-order complexity.

5 Implementation Details of SPFuzz

Based on the system call pattern coverage, we implement the first guided fuzzer, SPFuzz, that satisfies all the following design goals simultaneously:

- Feasibility: SPFuzz can support almost all COTS binaries that use system calls because it does not require rewriting (statically or dynamically) the target program.
- Efficiency: SPFuzz is as efficient as conventional instrumentation-based guided fuzzers since it does not require complex simulation.
- Effectiveness: SPFuzz can find magnitudes more code paths than black-box fuzzers
- Hardware Independence: SPFuzz does not require dedicated hardware, such as Intel-PT. Therefore, it can be run on all hardware platforms.

We have implemented SPFuzz using AFL (Version 2.57b) as a foundation, to keep consistent with our evaluation baselines. Our implementation involves a kernel driver that captures the system calls of applications being fuzzed. The driver consists of 3k lines of C/C++ code. We have integrated AFL with the kernel driver and adjusted its code to incorporate the coverage of the system call pattern.

The main alteration we made to AFL was to the input prioritization function. AFL assesses the prioritization score (energy) of an input by taking into account the number of newly discovered basic block transition edges, its creation time, and its execution time. On the contrary, SPFuzz uses the same input prioritization function as AFL, except for one difference. Instead of calculating the number of newly discovered basic block transition edges, SPFuzz uses the number of newly discovered system call patterns (the hash values), as shown in Equation 3. Such an alteration is not easy since we need to change the inner feedback structure of AFL and handle the collecting issues (will be illustrated in Section 5.1) to ensure that all the design goals are satisfied.

To prepare for the fuzzing loop, SPFuzz will execute the seeds of the target program and collect the call traces of the

system. This is necessary to determine the appropriate system call parameters for generating system call patterns for the target program, as explained in Section 4.

To give a better evaluation with AFL++Nyx and AFL++QEMU, we also implemented SPFuzz++ based on AFL++ (Version 4.10c). SPFuzz++ adopts the same feedback mechanism as SPFuzz and only differs in the implementation base.

5.1 System Call Trace Collection

One of the key challenges to collecting system call sequences is that existing auditing frameworks, such as Linux Audit [51], Sysdig [24], eBPF [48], and LTTng [29], can randomly drop system call events during fuzzing. This is particularly problematic, as these frameworks will drop 90% of system calls during fuzzing, making it impossible to use the system call patterns [41]. Furthermore, we find that there are no straightforward solutions to fix the event-dropping problem. For example, we have tried to let the auditing frameworks write the system call sequences to a file in the in-memory file system and then read the file in the fuzzer. However, even if we use the fast in-memory file system, the auditing frameworks can still drop 90% of system call events.

Motivated by the related work [41], we adopt a distributed kernel buffer strategy for SPFuzz and SPFuzz++ that naturally provides isolation between processes. Instead of using a single buffer, we allocate separate kernel buffers for each fuzzing instance. This design will prevent data races that arise due to the centralized architecture of existing auditing frameworks. Additionally, our distributed kernel buffer will allow SPFuzz to handle the system call sequences of different fuzzing instances individually. This will enable us to have better control over the generation speed of system calls for each instance. For example, if one instance generates system calls too quickly, SPFuzz and SPFuzz++ can block that instance to prevent any system calls from being dropped without interference with other instances.

5.2 Kernel Buffer Allocation

We adopt a dynamic buffer allocation strategy for SPFuzz and SPFuzz++. At the beginning, we allocate one page (4KB) of memory for each fuzzing instance. When the kernel buffer is full, we automatically double the size of the buffer. We choose this dynamic allocation approach because different targets demand different buffer sizes, ranging from 1MB to 128MB in our experiments. This design ensures that SPFuzz and SPFuzz++ are memory efficient and do not consume too many resources, especially when there are many fuzzing instances.

6 Evaluation

We focus on evaluating whether system call coverage guides greybox fuzzers as effectively as conventional code coverage guided approaches. In particular, we answer the following research questions.

- RQ 1: Can SPFuzz and SPFuzz++ run more realistic applications without causing errors?
- RQ 2: Can SPFuzz and SPFuzz++ cover new code as effectively as conventional code coverage guided approaches?
- RQ 3: Can SPFuzz and SPFuzz++ find bugs as effectively as conventional approaches?
- RQ 4: How well can the system call coverage approximate code coverage?
- RQ 5: Is the system call pattern coverage a stable feedback?
- RQ 6: How effective are the long-short system call coverage and our parameter selection algorithm?
- RQ 7: How well can SPFuzz++ perform compared with the source-code instrumented AFL++ (which should be the upper bound)?

Due to space limits, we leave some extra figures (Figure 4) and tables (Tables 7-12) in our GitHub repo at *https://github.com/Nova-xiao/SPFuzz*.

6.1 Experiment Setup

Infrastructure: Our experiments were carried out on four servers with Ubuntu 20.04 x86-64 OS. All these servers are equipped with eight Intel Xeon Gold 5218R cores and 8 GB of memory. For each fuzzing target program, we started four fuzzing instances in parallel on one server and ran for 24 hours. We repeat our experiment 10 times and report the average and standard deviation.

Benchmark Selection: We design a benchmark with 29 applications that can cover common cases and worst cases for SPFuzz. We used the same 13 Linux real-world benchmarks from ZAFL [54] to make a fair comparison between SPFuzz and baselines in common scenarios, including the open-source `readelf`, `tcpdump`, `bsdtar`, `cert-basic`, `sfconvert`, `unrtf`, `jasper`, `clean_text` and the closed-source `nconvert`, `nvdisasm`, `pngout`, `unrar`, `idat64`. Besides, we also add 16 extra applications to further evaluate SPFuzz and SPFuzz++. To evaluate the performance of SPFuzz and SPFuzz++ in the worst scenario, we select applications in the SPECCPU 17 Integer benchmark [44], which are supposed to use fewer I/O system calls than real-world applications and expected to make SPFuzz less effective.

Specifically, we selected five applications from the ten unique applications in SPECCPU 17 Integer, namely `perlbench_r`, `mcf_r`, `omnetpp_r`, `cpuxalan_r`, `x264_r`. We ignored "cpugcc" since it requires structured inputs, the efforts on the mutations are orthogonal to our work. We also ignored three AI algorithms in SPECCPU 17 Integer because they are not suitable for greybox fuzzing. Lastly, we ignored "xz" since it takes a long time to execute and none of our baselines can finish fuzzing effectively. Furthermore, besides the five closed-source applications used by ZAFL, we also included six Linux COTS closed-source applications, `cuobjdump`, `acroread`, `rar`, `kzip`, `zipmix`, and `lzturbo`, to evaluate whether SPFuzz and SPFuzz++ can find bugs as effectively as its binary-only baselines. To evaluate the performance of SPFuzz and SPFuzz++ on complex binaries, we select five C/C++ benchmarks that are not evaluated by previous works from OSSFuzz [57], namely `nginx`, `lua`, `gzip`, `pdftotext`, `z3`, whose input file formats and characteristics differ from ZAFL benchmarks and the binary sizes are at least 1M. The description and details of these applications can be found in Table 12 in our GitHub repo. To give a sanity check for SPC feedback, we also included the MAGMA [37] benchmark in RQ 7.

Baseline Selection: We chose baselines to thoroughly and fairly compare system call coverage with other conventional feedback mechanisms. We chose representative open-source approaches from all three categories of greybox binary fuzzing techniques. For static rewriting-based techniques, we chose ZAFL [54], which has the best feasibility for COTS binaries. For Intel-PT-based fuzzers, our first baseline is AFL++Nyx [60], the state-of-the-art approach that collects coverage feedback with Intel-PT. Besides, we also chose to use the PT-fuzzer [5], which is one of the most widely used fuzzers that is based on the vanilla version of AFL. For emulator-based fuzzers, we chose AFL-QEMU [4] and AFL++QEMU [12]. Specifically, we choose the Biondo version of AFL-QEMU [23], which contains the most recent engineering improvements. Lastly, we also include StochFuzz [68], which is based on static rewriting techniques but combines a dynamic design to make sure the approach should be sound.

SPFuzz++, AFL++QEMU and AFL++Nyx are built on AFL++, while SPFuzz and other baselines are built on AFL. Therefore, SPFuzz++, AFL++QEMU and AFL++Nyx use the same mutation strategy and share a large amount of infrastructure. The only main difference between them is the feedback strategy. This also applies to SPFuzz and AFL-based baselines.

Configuration: Since ZAFL did not publish its seeds, we choose the protocol of RetroWrite to collect initial seeds from the test cases in target source codes or releases [30] except for `nginx` and `z3` in our experiments, which lack test cases. For `nginx`, we collect requests from the Phoronix Test Suite [49], and for `z3`, we use scripts from the official z3Test repository [55].

For all baselines, we try to expand the timeout and memory parameters to 5000 ms and 8GB at most if they cannot fuzz successfully under default configurations. If they still crash during the calibration stage or are stuck on one execution for a long time(>6h), we will report them as failures.

Following Klee et al's [43] recommendation, we compute Mann-Whitney U-tests with a 0.05 significance level for important metrics such as crash numbers, coverage, and speed, and report them in the tables of the following RQs.

6.2 RQ 1: Feasibility

We report whether each application can run smoothly with SPFuzz and different baselines in Table 1 (closed-source applications). For each fuzzer, we mark the failed applications with ✗. Our evaluation shows that SPFuzz and SPFuzz++ can support more applications than other binary-only approaches.

We further investigated the reasons for the failures and discovered that they were caused by technical reasons intrinsic to different techniques. ZAFL and StochFuzz failed to fuzz six and 11 applications, respectively. The first reason for failures is that these static rewriters are unable to precisely analyze the pointers. For example, ZAFL fails to analyze the stack pointers for `z3` in our experiments, similar problems have been reported for other programs on the repo of ZAFL [52]. The second reason for failures is that these static rewriters may break the correctness of the program. For example, `nvdisasm` crashes with normal inputs after being instrumented by ZAFL, which breaks an indirect jump of the program. Note that we are using a newer version of `nvdisasm` (12.0 in our experiments), which results in an instrumentation performance different from the ZAFL paper [54].

AFL-QEMU and AFL++Nyx failed to fuzz seven applications in the beginning because they rely on QEMU, which cannot support AVX instructions. Therefore, we changed the compilation to the very outdated core2 architecture, as the AFL-QEMU document suggests [12]. AFL-QEMU works well after these changes, but AFL++Nyx itself still crashes when fuzzing three of them since it uses an even older QEMU version (4.20). We mark the benchmarks with special compilation settings with underlines in Table 1 and Table 2.

Finally, PT-fuzzer failed to fuzz 10 applications because Intel-PT generated too many tracing events that overflowed the event buffer of PT-fuzzer. We notice that simply increasing the event buffer of the PT-fuzzer cannot address the buffer overflow problem. In our experiment, we have increased the event buffer from the default size of 128MB to the maximum memory size of our hardware platform (8GB). However, we still encounter the buffer overflow problem. This result indicates that we need a more fundamental improvement over the PT-fuzzer to make it more robust.

6.3 RQ 2: Code Coverage and Fuzzing Speed

We use **branch coverage** [27, 65], a widely adopted metric in software testing, to evaluate whether SPFuzz and SPFuzz++ can discover new code as effectively as conventional approaches. It refers to the number of branches reached divided by the total number of branches in the codebase [69]. At the same time, we record the execution speed of all baselines. We conducted 24-hour fuzzing ten times on each open-source application using these tools and calculated the average coverage data. For SPFuzz, SPFuzz++ and other binary-only baselines, we saved the executed seeds and replayed them offline using `gcov` instrumented applications to collect coverage data. The results are shown in Table 2, which shows the average coverage, speed, and standard deviation for ten iterations of the experiment. We omit closed-source applications in this section due to the difficulty of collecting branch coverage data for them.

Our evaluation shows that system call pattern coverage can guide fuzzing almost as effectively as conventional approaches. Compared to other baselines, SPFuzz achieves higher coverage than AFL-QEMU and PT-Fuzzer because the execution speed of these two baselines is much lower than SPFuzz. For example, AFL-QEMU runs `nginx` 24 times slower than SPFuzz, leading to a much smaller coverage number. This is consistent with our insight that improving execution speed can improve code coverage in practice.

On benchmarks that are feasible to ZAFL, the difference in the branch coverage achieved by ZAFL and SPFuzz is less than 0.1% on average. With respect to speed, SPFuzz is 2% slower than ZAFL on most benchmarks except `cert-basic` and `jasper`. Due to their simple inner logic and small basic blocks, the instrumentation-pruning and instrumentation-downgrading techniques adopted by ZAFL [54] are extraordinarily effective for these two applications. For StochFuzz, its speed is on average 8% less than SPFuzz, but its coverage is on average 0.1% higher due to its sound instrumentation-based feedback. Overall, SPFuzz has similar performance as static rewriting-based approaches on most benchmarks, while having higher feasibility.

Regarding SPFuzz++, it demonstrates superior coverage and speed compared to all baseline tools on 11 out of 18 benchmarks. On the other benchmarks, SPFuzz++ maintains coverage that is not statistically significantly different from the best baseline tool. The exceptions to this performance are `cert-basic` and `jasper`. ZAFL runs faster on `cert-basic` and `jasper` due to specific optimizations mentioned earlier, while StochFuzz is faster on `readelf` because its heuristic algorithm is more effective for this particular target.

Specifically, SPFuzz++ achieves equal or better coverage than both AFL++QEMU and AFL++Nyx across all benchmarks, which can be attributed to its enhanced speed. Notably, SPFuzz++ outperforms AFL++Nyx on `nginx`, a benchmark that relies more heavily on syscalls.

Coverage Growth Speed. To demonstrate the details of how fast SPFuzz and SPFuzz++ can find new code, we have included a graph in Figure 2 that shows the growth in branch coverage over time. The y-axis represents the number of covered branches, while the x-axis displays the execution time.

As shown in Figure 2, although the number of new branches discovered by SPFuzz grows slower than ZAFL when it is feasible, it still grows correspondingly. The growth speed of the coverage of SPFuzz is also similar to other baselines. For SPFuzz++, its coverage grows faster than most baselines on all benchmarks except `tcpdump` and `lua`, where it starts to show a significant advantage in the latter half. This result indicates that system call path coverage can be an effective approximation for code coverage feedback when instrumentation is not possible.

6.4 RQ 3: Bug Finding

To evaluate the bug-finding capacity of SPFuzz and SPFuzz++, we report the number of triaged crashes that we have discovered in our open-source applications for SPFuzz, SPFuzz++ and its binary-only baselines. In addition, we further evaluate whether SPFuzz and SPFuzz++ can find bugs in realistic binaries in the seven COTS applications. For COTS applications, we directly report the crashes discovered in fuzzing. For open-source applications, we use ASAN [59] to detect crashes. However, instead of directly fuzzing the ASAN-sanitized binaries, we choose to do our experiment in a post-mortem way: when the fuzzing campaigns finish, we run all the saved test cases (i.e., queued, hanged, and crashed ones) with ASAN-sanitized binaries and add the found crashes to our results. This is because AFL-QEMU [12], AFL++QEMU [12] and ZAFL have problems in fuzzing ASAN-sanitized binaries. Nevertheless, our evaluation can still faithfully evaluate how well can SPFuzz, SPFuzz++ and baselines find bugs.

Crash Triage. Instead of counting the auto-allocated crash ids, we follow the existing "fuzzy stack hashing" methodology to triage all these crashes [1, 43]. We first collect stack traces and errors with ASAN (for open-source) and GNU Debugger (for closed-source), then hash each crash with the addresses and errors in the top 6 entries of their stack traces. Table 1 includes the names of these applications, together with the number of triaged crashes we found.

ZAFL and PTFuzzer do not support fuzzing `acroread` because these fuzzers can only fuzz 64-bit applications, while `acroread` only has a 32-bit binary. Thus, we leave the corresponding entries as N/A.

Our results show that SPFuzz and SPFuzz++ were effective in identifying crashes. Specifically, SPFuzz found more crashes than AFL-QEMU and PTFuzzer in all applications. It also found more crashes than ZAFL, AFL++Nyx, StochFuzz in five to seven applications. As for SPFuzz++, it found identical or more crashes than all baselines. In particular, SP-

Table 1: The triaged crash numbers discovered in 24 hours. Column O/C indicates whether the application is open-source (O) or closed-source (C). ✗ indicates a failed execution. N/A means that the fuzzer does not support running the application. We report the average (avg) and standard deviation (std) of ten rounds of experiments in the form avg | std. We mark the data with $^+$ or $^-$ if SPFuzz++ has a statistically distinguishable advantage or disadvantage over it with MWU test p-value less than 0.05, respectively. We mark the benchmarks with special compilation settings with underlines. We also mark the item that shows a statistically distinguishable advantage over most other baselines with **bold**.

O/C	Benchmark	SPFuzz++	AFL++QEMU	SPFuzz	AFL-QEMU	ZAFL	PTFuzzer	AFL++Nyx	StochFuzz
O	nginx	1 | 0	1 | 0	1 | 0	1 | 0	✗	✗	1 | 0	✗
O	lua	2 | 1	2 | 1	2 | 1	0^+ | 0	1^+ | 1	✗	2 | 1	✗
O	gzip	5 | 1	5 | 1	5 | 2	3^+ | 1	✗	0^+ | 0	4 | 1	3^+ | 1
O	pdftotext	**35** | **3**	24^+ | 4	32^+ | 6	6^+ | 3	22^+ | 11	0^+ | 0	33 | 12	17^+ | 5
O	z3	**30** | **11**	2^+ | 1	28^+ | 10	0^+ | 0	✗	0^+ | 0	0^+ | 0	✗
O	readelf	1 | 0	1 | 0	1 | 0	1 | 0	1 | 0	1 | 0	1 | 0	
O	tcpdump	16 | 4	15 | 2	12^+ | 2	5^+ | 1	14 | 2	8^+ | 8	16 | 16	✗
O	bsdtar	8 | 2	6^+ | 1	7^+ | 2	1^+ | 0	8 | 3	0^+ | 0	8 | 4	7 | 1
O	cert-basic	10 | 1	10 | 1	8^+ | 1	4^+ | 1	10 | 2	✗	8^+ | 1	2^+ | 0
O	sfconvert	2 | 0	2 | 0	2 | 0	1^+ | 0	2 | 0	✗	2 | 0	0^+ | 0
O	unrtf	16 | 2	15 | 1	15^+ | 2	7^+ | 1	16 | 3	✗	12^+ | 2	10^+ | 1
O	jasper	3 | 1	3 | 0	3 | 0	2^+ | 0	3 | 0	1^+ | 0	3 | 0	3 | 0
O	clean_text	**12** | **2**	9^+ | 1	6^+ | 1	1^+ | 0	5^+ | 1	✗	11 | 1	3^+ | 2
O	perlbench_r	1 | 0	1 | 0	1 | 0	0^+ | 0	1 | 0	0^+ | 0	0^+ | 0	✗
O	mcf_r	18 | 3	17 | 1	16^+ | 2	13^+ | 1	7^+ | 5	1^+ | 0	**28**$^-$ | **1**	3^+ | 1
O	omnetpp_r	1 | 0	1 | 0	1 | 0	1 | 0	1 | 0	1 | 0	✗	✗
O	cpuxalan_r	1 | 0	1 | 0	1 | 0	1 | 0	✗	✗	✗	✗
O	x264_r	1 | 1	1 | 0	1 | 1	0^+ | 0	0^+ | 0	✗	✗	✗
C	nconvert	**5** | **0**	3^+ | 0	3^+ | 0	0^+ | 0	2^+ | 0	0^+ | 0	0^+ | 0	0^+ | 0
C	nvdisasm	**128** | **5**	99^+ | 12	121^+ | 1	13^+ | 9	✗	0^+ | 0	42^+ | 3	67^+ | 23
C	cuobjdump	**151** | **38**	150 | 35	147^+ | 36	136^+ | 28	150 | 22	0^+ | 0	146 | 43	50^+ | 2
C	acroread	3 | 1	N/A	N/A	N/A	N/A	✗	✗	✗
C	pngout	4 | 1	4 | 1	4 | 1	0^+ | 0	✗	✗	✗	✗
C	unrar	10 | 1	9^+ | 1	9^+ | 1	10 | 1	10 | 1	✗	10 | 1	9^+ | 0
C	rar	3 | 2	3 | 1	3 | 2	3 | 1	3 | 0	✗	2^+ | 1	2^+ | 0
C	idat64	1 | 0	1 | 0	1 | 0	1 | 0	1 | 0	✗	1 | 0	✗
C	kzip	4 | 1	4 | 1	4 | 1	3^+ | 0	**5**$^-$ | **1**	1^+ | 0	4 | 1	3^+ | 0
C	zipmix	1 | 0	1 | 0	1 | 0	1 | 0	1 | 0	1 | 0	1 | 0	1 | 0
C	lzturbo	2 | 1	2 | 0	2 | 0	2 | 0	2 | 1	✗	2 | 1	1^+ | 0

Fuzzand SPFuzz++ can fuzz all applications without causing errors, while ZAFL, AFL-QEMU, AFL++Nyx and StochFuzz still crash or do not support a subset of applications, or require special compilation settings.

We reported the valid crashes **discovered by SPFuzz** to the developers of these applications. Developers have confirmed 13 of them. In particular, SPFuzz has obtained six new CVEs. Among the six CVEs, we note that ZAFL cannot find a CVE in `nvdisasm` since the transformed binary crashed with normal inputs. Furthermore, AFL-QEMU is too slow to run in this case. Holistically, our evaluation shows that a guided fuzzer based on system call pattern coverage is effective in finding new bugs in COTS binaries.

6.5 RQ 4: Approximating Code Coverage

The key insight of this paper is that system call pattern coverage can approximate code coverage. In other words, we assume that the system call patterns are statistically correlated with the code coverage. For example, a higher system call pattern coverage indicates a higher code coverage. In this section, we aim to evaluate whether our insight is valid in our experiments.

In order to validate our insight, we employ two types of statistical correlation measure, namely Pearson's r [56] and Kendall's τ [15]. Pearson's r assesses the linear relationship between two variables, while Kendall's τ evaluates the positive correlation between two variables, even in the absence of a linear relationship. Both Pearson's r and Kendall's τ range from -1 to 1, where negative values indicate a negative relationship between the variables and positive values indicate a positive relationship. To interpret the values of Pearson's r and Kendall's τ, we follow the standard approach, where a range of 0.4 - 0.6 signifies a moderate relationship between the variables, and a range of 0.6 - 1 indicates a strong relationship [16].

We employ Pearson's r and Kendall's τ to calculate the correlation between system call pattern coverage and code coverage through the following procedure. Initially, we randomly generated 400 pairs of inputs (i_0, i_1) for each open-source application and executed them individually. These inputs, i_0 and i_1, are produced during the fuzzing process described

Table 2: The averages (left) and standard deviations (right) of the branch coverage and speed on different baselines after 24-hour running. ✗ represents failures or jams. We report the average (avg) and standard deviation (std) of ten rounds of experiments in the form avg|std. We mark the data with $^+$ or $^-$ if SPFuzz++ has a statistically distinguishable advantage or disadvantage over it with MWU test p-value less than 0.05, respectively. We also mark the item that shows a statistically distinguishable advantage over all other baselines with **bold**. We mark the benchmarks with special compilation settings with underlines.

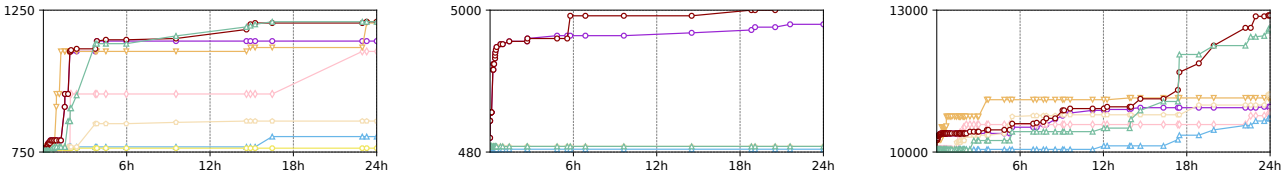
Benchmark	**SPFuzz++** coverage	**SPFuzz++** speed	**AFL++QEMU** coverage	**AFL++QEMU** speed	**AFL++Nyx** coverage	**AFL++Nyx** speed	**StochFuzz** coverage	**StochFuzz** speed
nginx	**21.1%** | **0.2%**	**380** | **45**	1.9%$^+$ | 0.1%	88$^+$ | 31	1.9%$^+$ | 0.1%	339$^+$ | 10	✗	✗
lua	**89.2%** | **3.0%**	**145** | **31**	87.9%$^+$ | 2.9%	129$^+$ | 44	87.7%$^+$ | 1.9%	130$^+$ | 61	✗	✗
gzip	**33.8%** | **0.1%**	**196** | **4**	33.0%$^+$ | 0.1%	178$^+$ | 4	26.0%$^+$ | 0.6%	166$^+$ | 33	25.8%$^+$ | 0.0%	65$^+$ | 45
pdftotext	29.9% | 0.1%	188 | 28	29.9% | 0.2%	196 | 17	22.2%$^+$ | 0.2%	44$^+$ | 3	29.6%$^+$ | 0.0%	192 | 18
z3	**8.3%** | **0.1%**	**191** | **72**	7.7%$^+$ | 0.1%	20$^+$ | 1	7.6%$^+$ | 0.1%	14$^+$ | 7	✗	✗
readelf	12.3% | 0.1%	290 | 19	11.0%$^+$ | 0.1%	78$^+$ | 5	12.3% | 0.1%	211$^+$ | 31	9.9%$^+$ | 0.9%	**383**$^-$ | **19**
tcpdump	**58.0%** | **1.2%**	**180** | **32**	57.9% | 0.8%	158$^+$ | 35	50.7%$^+$ | 0.2%	107$^+$ | 11	✗	✗
bsdtar	**7.9%** | **1.0%**	**178** | **39**	7.0%$^+$ | 0.4%	123$^+$ | 25	7.2%$^+$ | 1.0%	160 | 116	7.0%$^+$ | 0.1%	144$^+$ | 11
cert-basic	66.2% | 0.1%	1154 | 23	66.2% | 0.1%	1142 | 41	22.2%$^+$ | 0.2%	948$^+$ | 35	64.4%$^+$ | 0.0%	86$^+$ | 16
sfconvert	**39.9%** | **0.1%**	**480** | **44**	37.9%$^+$ | 0.1%	51$^+$ | 5	38.0%$^+$ | 0.0%	55$^+$ | 22	36.5%$^+$ | 0.0%	14$^+$ | 12
unrtf	1.2% | 0.0%	1045 | 22	1.2% | 0.0%	1008 | 30	1.1%$^+$ | 0.0%	1021 | 436	1.1%$^+$ | 0.0%	4$^+$ | 1
jasper	**30.3%** | **0.3%**	279 | 36	30.1% | 0.2%	140$^+$ | 27	29.1%$^+$ | 0.0%	129$^+$ | 64	29.1%$^+$ | 0.0%	87$^+$ | 15
clean_text	26.9% | 0.1%	156 | 31	26.9% | 0.1%	148 | 49	26.8% | 0.2%	80$^+$ | 21	19.9% $^+$ | 0.1%	16$^+$ | 2
perlbench_r	**18.3%** | **0.5%**	246 | 41	17.0%$^+$ | 0.4%	192$^+$ | 33	17.1%$^+$ | 0.3%	230$^+$ | 91	✗	✗
mcf_r	75.3% | 0.2%	248 | 20	75.3% | 0.0%	256 | 46	73.1%$^+$ | 0.0%	33$^+$ | 5	73.1%$^+$ | 0.0%	196$^+$ | 14
omnetpp_r	8.9% | 0.2%	391 | 56	7.1%$^+$ | 0.1%	149$^+$ | 32	✗	✗	✗	✗
cpuxalan_r	**15.6%** | **0.1%**	**189** | **29**	9.1%$^+$ | 0.1%	71$^+$ | 13	✗	✗	✗	✗
x264_r	**12.6%** | **0.2%**	**236** | **41**	11.6%$^+$ | 0.0%	190$^+$ | 35	✗	✗	✗	✗

Benchmark	**SPFuzz** coverage	**SPFuzz** speed	**AFL-QEMU** coverage	**AFL-QEMU** speed	**ZAFL** coverage	**ZAFL** speed	**PTfuzzer** coverage	**PTfuzzer** speed
nginx	18.5%$^+$ | 0.0%	286$^+$ | 52	1.0%$^+$ | 0.1%	12$^+$ | 4	✗	✗	✗	✗
lua	83.2%$^+$ | 3.4%	134$^+$ | 26	83.0%$^+$ | 2.8%	116$^+$ | 85	81.0%$^+$ | 2.0%	109$^+$ | 8	✗	✗
gzip	26.0%$^+$ | 0.1%	125$^+$ | 4	25.7%$^+$ | 0.1%	91$^+$ | 3	✗	✗	24.6%$^+$ | 0.6%	101$^+$ | 29
pdftotext	28.4%$^+$ | 0.0%	136$^+$ | 20	21.0%$^+$ | 0.1%	32$^+$ | 15	29.8%$^-$ | 0.1%	150$^+$ | 6	26.6%$^+$ | 0.0%	111$^+$ | 68
z3	8.0%$^+$ | 0.1%	187$^+$ | 65	7.7%$^+$ | 0.1%	15$^+$ | 5	✗	✗	7.4%$^+$ | 1.6%	8$^+$ | 3
readelf	10.5%$^+$ | 0.1%	241$^+$ | 20	10.1%$^+$ | 0.1%	12$^+$ | 1	11.5%$^+$ | 0.2%	283 | 6	7.5%$^+$ | 0.4%	111$^+$ | 68
tcpdump	49.1%$^+$ | 0.2%	122$^+$ | 23	47.8%$^+$ | 0.4%	10$^+$ | 2	50.1%$^+$ | 0.7%	126$^+$ | 26	48.4%$^+$ | 1.5%	101$^+$ | 8
bsdtar	7.2%$^+$ | 0.8%	149$^+$ | 31	5.0%$^+$ | 0.0%	39$^+$ | 7	7.2%$^+$ | 1.0%	169 | 44	2.3%$^+$ | 0.0%	2$^+$ | 1
cert-basic	64.4%$^+$ | 0.0%	62$^+$ | 11	64.4%$^+$ | 0.0%	42$^+$ | 4	64.5%$^+$ | 0.2%	**1460**$^-$ | **343**	✗	✗
sfconvert	38.0%$^+$ | 0.0%	461$^+$ | 13	36.5%$^+$ | 0.1%	9$^+$ | 2	38.0%$^+$ | 0.0%	126$^+$ | 32	✗	✗
unrtf	1.1% | 0.0%	23$^+$ | 12	1.1% | 0.0%	1$^+$ | 0	1.1% | 0.0%	6$^+$ | 2	✗	✗
jasper	29.1%$^+$ | 0.2%	158$^+$ | 23	29.0%$^+$ | 0.0%	4$^+$ | 1	29.1%$^+$ | 0.2%	**1041**$^-$ | **111**	28.1%$^+$ | 0.0%	10$^+$ | 1
clean_text	21.8%$^+$ | 0.1%	63$^+$ | 5	21.6% $^+$ | 0.0%	35 $^+$ | 7	20.4% $^+$ | 0.1%	128$^+$ | 12	✗	✗
perlbench_r	17.8%$^+$ | 0.6%	225$^+$ | 39	12.0%$^+$ | 0.2%	129$^+$ | 12	17.8%$^+$ | 0.1%	238 | 40	14.6%$^+$ | 0.3%	8$^+$ | 5
mcf_r	73.1%$^+$ | 0.0%	228 | 21	73.1%$^+$ | 0.0%	110$^+$ | 60	73.1%$^+$ | 0.0%	148$^+$ | 13	✗	✗
omnetpp_r	8.3%$^+$ | 0.1%	368 | 20	6.5%$^+$ | 0.1%	57$^+$ | 3	8.9% | 0.1%	291$^+$ | 43	7.0%$^+$ | 0.1%	55$^+$ | 8
cpuxalan_r	15.2%$^+$ | 0.0%	167$^+$ | 12	8.9%$^+$ | 0.1%	6$^+$ | 1	✗	✗	15.2%$^+$ | 0.1%	109$^+$ | 21
x264_r	11.5%$^+$ | 0.1%	123$^+$ | 24	1.1%$^+$ | 0.0%	3$^+$ | 1	✗	✗	11.5%$^+$ | 0.1%	2$^+$ | 1

in Section 6.3. Subsequently, we compute the differences in system call patterns and code coverage between running i_1 and i_0, denoted as Δsys and $\Delta code$, respectively. Finally, we use Pearson's r and Kendall's τ to quantify the relationship between Δsys and $\Delta code$.

Due to the page limit, we put the detailed result in Table 9 in our GitHub repo. All reported values have a p value below 0.01, indicating their statistical significance. Our evaluation demonstrates that system call pattern coverage can serve as an approximation for code coverage. Out of the 15 applications, there exists a significant linear relationship between system call pattern coverage and code coverage ($r > 0.6$). Only three applications exhibit r values below 0.4. Nonetheless, all of them exhibit Kendall values (τ) higher than 0.4. This indicates that system call pattern coverage exhibits a moderate to strong positive correlation with code coverage across all applications in our experiment.

Moreover, for every open-source benchmark, we collect the seeds saved by AFL in 24 hours and feed them to SPFuzz to test if their system call feedbacks are different. Results show that SPFuzz can distinguish over 95% test cases on every benchmark. For SPFuzz++, the results are identical since it utilizes the same feedback as SPFuzz. This indicates that system call pattern coverage is enough to approximate traditional code coverage.



(a) pdftotext (b) nginx (ZAFL,PTFuzzer,StochFuzz fail) (c) tcpdump (StochFuzz fails)

Figure 2: The covered branches after 24-hour running. In every subfigure, the x-axis represents the execution time, and the y-axis represents the number of reached branches. The purple line with circles represents SPFuzz, the red line with circles represents SPFuzz++, the cyan line with triangles represents AFL-QEMU, the green line with triangles represents AFL++QEMU, the orange line with inverted triangles represents ZAFL, the pink line with diamonds represents PTfuzzer, the brown line with pentagons represents AFL++Nyx, and the yellow line with octagons represents StochFuzz. We present only three of them due to space limits, and others can be found in Figure 4 in our Github repo. Note that some lines can be missing in specific figures since the corresponding baselines fail to work.

6.6 RQ 5: Stability

Stability is a measure of how effectively a feedback mechanism can differentiate between different inputs [10], which is crucial for a feedback method. A feedback method is considered stable if it produces the same output for the same inputs and different outputs for different inputs. To evaluate the stability of SPFuzz and SPFuzz++, we follow the standard procedure for measuring the stability of a feedback method [6]: for each input in the initial corpus, we repeatedly run the fuzzer (eight times in our implementation, the same as AFL) and record the feedback. We then calculate the percentage of the feedback bitmap that remains unchanged for identical inputs. Next, we evaluate how well the fuzzer can differentiate between different inputs by comparing the editing distance of the system call sequences for the same input to that of different inputs. The results of stability are presented in Table 3, while the results of editing distance of the system call sequences are shown in Table 10 in our GitHub repo due to page limits.

Our evaluation demonstrates that the system call pattern serves as a reliable feedback method. SPFuzz and SPFuzz++ both achieve a stability rate of 100% across all applications. In other words, when provided with two identical inputs, the system will generate identical system call sequences for all applications. This is akin to the feedback obtained from code coverage. The editing distances between system call sequences of identical inputs are found to be 0. Conversely, the editing distances for different inputs are non-zero values. This outcome indicates that SPFuzz and SPFuzz++ can effectively differentiate between different inputs, thereby ensuring an effective feedback mechanism.

6.7 RQ 6: Effectiveness of Optimizations

In this section, we evaluate the effectiveness of the optimizations we made in Section 4.

Table 3: The impacts of parameter selection when applying system call trace guided fuzzing. In the context of system call trace guided fuzzing, path refers to *SPC* found by the fuzzer.

Benchmark	New Path(*SPC*)			Stability		
	No	SPFuzz & SPFuzz++	All	No	SPFuzz & SPFuzz++	All
nginx	✗	✓	✓	100.00%	100.00%	0.44%
lua	✗	✓	✓	100.00%	100.00%	0.42%
gzip	✓	✓	✓	100.00%	100.00%	0.17%
pdftotext	✓	✓	✓	100.00%	100.00%	0.84%
z3	✓	✓	✓	100.00%	100.00%	2.88%
tcpdump	✓	✓	✓	100.00%	100.00%	0.48%
readelf	✓	✓	✓	100.00%	100.00%	0.15%
bsdtar	✓	✓	✓	100.00%	100.00%	0.61%
cert-basic	✗	✓	✓	100.00%	100.00%	3.64%
sfconvert	✓	✓	✓	100.00%	100.00%	0.11%
unrtf	✗	✓	✓	100.00%	100.00%	2.70%
jasper	✓	✓	✓	100.00%	100.00%	0.14%
perlbench_r	✓	✓	✓	100.00%	100.00%	0.03%
mcf_r	✗	✓	✓	100.00%	100.00%	0.23%
omnetpp_r	✗	✓	✓	100.00%	100.00%	0.28%
cpuxalan_r	✓	✓	✓	100.00%	100.00%	3.75%
x264_r	✓	✓	✓	100.00%	100.00%	0.88%

Time to get θ: In order to evaluate the efficiency of the algorithm to get θ in Section 4, we have recorded the time from prompting with LLMs to the time we get θ for every benchmark in our experiments. We repeat the experiments for 10 times and calculate average time costs and the standard deviations. We show the average and standard deviations of the time to arrive at θ in Table 11 (in our GitHub repo). For all benchmarks, the time to arrive at θ is less than five minutes, which is acceptable since the reasonable and widely adopted time of a fuzzing campaign should be at least 24 hours [43, 46, 65].

Long-short system call pattern coverage: In order to evaluate the effects of long-short system call pattern coverage, we compare the branch coverage by solely using system call sequences of length θ (SPC_θ), 2θ ($SPC_{2\theta}$), and the complete

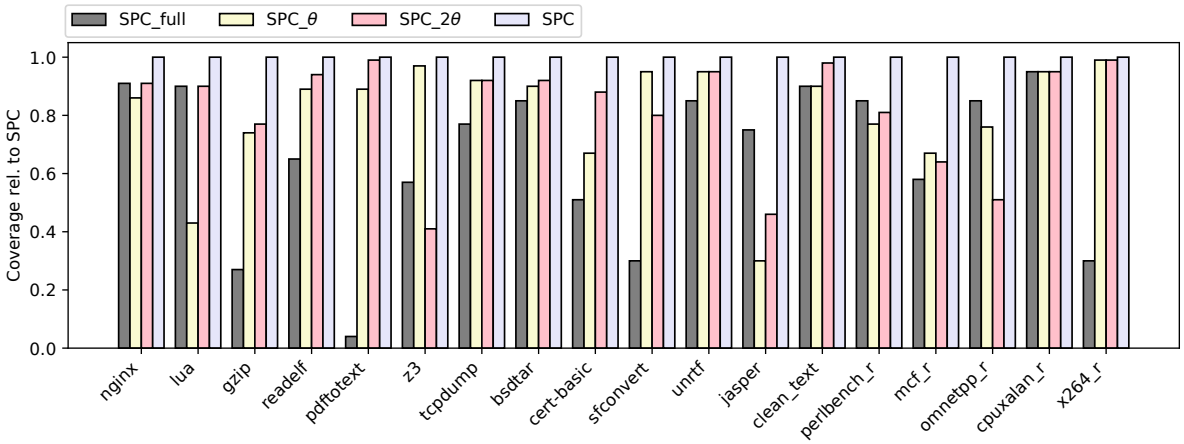


Figure 3: The comparison of branch coverage under different syscall guidance. We normalize the coverage data by SPFuzz as one.

system call sequences (SPC_{full}) with SPFuzz (*SPC*). We use the same benchmarks as discussed in Section 6.3 and present the coverage data for the 18 open-source applications in Figure 3. In this figure, we have standardized the coverage using the combined *SPC* as 1.

Our experimental results demonstrate that the utilization of the combined *SPC* leads to an improvement in branch coverage compared to SPC_{θ}, $SPC_{2\theta}$, and SPC_{full} by a maximum of 238%, 243%, and 2641%, respectively. This finding highlights the effectiveness of incorporating long-short system call pattern coverage into the fuzzing process. This conclusion also applies to SPFuzz++ since it shares the same feedback mechanism with SPFuzz. The new mutation and seed selection algorithms of AFL++ are orthogonal with the changes in feedback information.

Parameter selection: In order to demonstrate the effectiveness of our parameter selection algorithm, we conducted two baseline experiments. In the first baseline, we completely omitted the parameter selection algorithm, while in the second baseline, we used all available parameters. Our focus was on two specific metrics: the number of new paths (i.e. new *SPC*s for SPFuzz and SPFuzz++) discovered in the initial round of fuzzing, and the stability of the feedback data. In the context of fuzzing, the initial round refers to the first set of mutations applied to the initial input seeds. If no new paths are discovered in this round, an error indicator will be displayed on the AFL or AFL++ panel, indicating that the fuzzer is generally unable to explore further [32, 65]. During the initial round, all mutation methods are applied to all initial input seeds. Therefore, the absence of new paths in this round suggests that the feedback mechanism is unable to identify different paths.

Table 3 demonstrates that the system call coverage patterns without parameters do not accurately represent the inner logic of benchmarks like `cert-basic` when parameter recording is not used. As a consequence, no new paths are discovered, rendering fuzzing ineffective. In contrast, recording all parameters can result in the generation of numerous new paths. However, randomly modifying parameters can have a detrimental effect on stability by contaminating feedback information. Consequently, these brute force methods are not applicable, and the parameter selection method of SPFuzz and SPFuzz++ is a necessary approach that is sensitive and stable enough to facilitate effective greybox fuzzing.

Furthermore, we have identified that our parameter selection algorithm has the potential to improve fuzzing efficiency compared to the approach of recording all parameters. The results presented in Table 7 (in our GitHub repo) illustrate that our algorithm can increase the speed of fuzzing by a factor of up to 45. This improvement can be attributed to the reduction in the number of parameters that need to be stored in the kernel buffer, leading to a decrease in the overhead associated with extending the kernel buffers. In summary, our evaluation validates the effectiveness of our parameter selection algorithm in enhancing system call pattern coverage-guided fuzzing.

6.8 RQ 7: Sanity Check

To give a sanity check for the SPC feedback mechanism, we choose to evaluate the performance of SPFuzz++ and AFL++ [32] on the MAGMA benchmark [37]. We follow the former evaluation protocol to conduct experiments under the default configuration of MAGMA. AFL++ and SPFuzz++ are feasible on all binaries except `php-unserialize`, whose initial corpus provided by MAGMA contains no available seeds (all caused crashes). We have reported this problem to the MAGMA maintainer and removed it from Table 4.

As Table 4 shows, SPFuzz++ achieves identical coverage with AFL++ on six benchmarks since it gains similar speeds, and the SPC feedback also works well on these targets. The coverage difference of SPFuzz++ and AFL++ is

Table 4: The averages (left) and standard deviations (right) of the branch coverage and speed of the source-code instrumented AFL++ and SPFuzz++ after 24-hour running. We report the average (avg) and standard deviation (std) of ten rounds of experiments in the form avg|std. We mark the data with **bold** if it has a statistically distinguishable advantage with MWU test p-value less than 0.05.

Benchmark	**AFL++** coverage	**AFL++** speed	**SPFuzz++** coverage	**SPFuzz++** speed
libpng_read_fuzzer	**12.4%** | **0.1%**	**1292** | **301**	12.2% | 0.1%	766 | 135
sndfile_fuzzer	**17.5%** | **0.1%**	**1148** | **189**	16.3% | 0.1%	234 | 41
tiff_read_rgba_fuzzer	**6.6%** | **0.1%**	**1138** | **176**	6.4% | 0.1%	435 | 45
tiffcp	**44.8%** | **0.8%**	**403** | **96**	34.9% | 1.1%	221 | 40
xml_read_memory_fuzzer	**14.7%** | **0.2%**	**939** | **135**	14.3% | 0.2%	354 | 61
xmllint	**17.9%** | **0.2%**	257 | 76	14.4% | 0.2%	272 | 44
lua	**94.3%** | **2.3%**	**266** | **45**	89.2% | 3.0%	145 | 31
openssl_asn1	4.5% | 0%	**45** | **2**	4.5% | 0%	36 | 4
openssl_asn1parse	1.9% | 0%	183 | 34	1.9% | 0%	195 | 53
openssl_bignum	2.0% | 0%	397 | 78	2.0% | 0%	388 | 56
openssl_server	18.3% | 0.1%	251 | 32	18.3% | 0.2%	239 | 54
openssl_client	5.0% | 0%	**76** | **5**	5.0% | 0%	68 | 8
openssl_x509	**12.6%** | **0.2%**	**355** | **17**	12.3% | 0.4%	258 | 30
php_json	6.3% | 0%	**96** | **4**	6.3% | 0.1%	74 | 5
php_exif	**6.8%** | **0.1%**	**95** | **4**	6.5% | 0.1%	81 | 5
php_parser	**14.2%** | **0.1%**	**94** | **4**	14.1% | 0.1%	75 | 10
pdf_fuzzer	**28.1%** | **0.1%**	**906** | **53**	27.9% | 0.1%	630 | 42
pdfimages	**44.5%** | **0.3%**	**45** | **9**	40.8% | 1.5%	41 | 16
pdftoppm	**32.4%** | **0.2%**	**26** | **10**	31.2% | 0.3%	22 | 8
sqlite3_fuzz	**5.9%** | **0.1%**	**1282** | **233**	5.8% | 0.1%	801 | 189

less than 1% on nine benchmarks, where SPFuzz++ still keeps a comparable speed. For the other six benchmarks, the coverage of AFL++ is 1.2% to 9.9% better than SPFuzz++ since AFL++ gains an advantage from the more accurate and faster source-code instrumentation. Specifically, AFL++ gains a tremendous advantage on `tiffcp`. Upon closer examination, we found that the branches in its *tif_tile.c* component contain nearly no syscalls, which is very rare since normal computation-intense codes should at least contain memory-related syscalls. Though affected by this extreme situation, SPFuzz++ is still capable of exploring the other parts of this target normally. Overall, compared to the theoretical upper bound, AFL++, SPFuzz++ performs similarly across most benchmarks, with significant deviations occurring only in rare cases.

7 Discussion

7.1 Runtime Overhead

In Section 6.3 we reported that ZAFL and SPFuzz have comparable fuzzing speeds on most benchmarks. This is interesting because SPFuzz is an instrumentation-free approach, which means that it can not instrument the forkservers into the target programs. Since forkserver provides a significant improvement over fuzzing speed, we aim to explain why the fuzzing speed of SPFuzz is comparable to ZAFL.

We argue that the reason for the efficiency of SPFuzz is avoiding the runtime overhead introduced by instrumented probes that collect coverage data. To prove this argument, we measure the percentage of fuzzing speed improvement of forkserver and the runtime overhead of instrumented probes in AFL.

To measure the overhead of instrumented probes, we first run the AFL-Dumb mode with instrumented binaries with the initial fuzzing seeds, then run the AFL-Dumb mode with uninstrumented binaries and compare their execution speeds. To measure the optimization effects of forkservers, we run the normal AFL mode and compare their speeds with the speeds of instrumented binaries under AFL-Dumb mode. Note that we only run the initial seeds in this experiment, thus the inputs are identical for AFL-Dumb and AFL, eliminating possible biases introduced by different inputs from mutations.

The results in Table 5 demonstrate that the overhead of SPFuzz can be better than the optimized instrumentation method, especially when the target binary is big or complicated. Overall, our results show that the overhead of instrumented probes is comparable to the optimization of forkservers. This result explains the performance data in Table 2. For applications, such as `mcf_r` and `omnetpp_r`, that have higher instrumentation overheads, the fuzzing speed of SPFuzz is higher than ZAFL. Otherwise, ZAFL has a higher or similar fuzzing speed.

Table 5: The profiling of how instrumentation and forkserver influence the actual execution speeds.

Benchmarks	Instrumentation Overhead	Forkserver Optimization
nginx	10%	42%
lua	33%	21%
gzip	16%	9%
pdftotext	53%	42%
z3	15%	6%
tcpdump	373%	224%
readelf	1%	17%
bsdtar	28%	60%
cert-basic	298%	325%
sfconvert	40%	47%
unrtf	84%	30%
jasper	14%	43%
clean_text	1%	1%
perlbench_r	83%	5%
mcf_r	28%	8%
omnetpp_r	60%	33%
cpuxalan_r	306%	2%
x264_r	84%	34%

7.2 Computational Intensive Applications

SPFuzz and SPFuzz++ face a notable challenge when it comes to computation-intensive applications, as they typically

have fewer calls compared to IO-intensive applications. However, our evaluation statistics show that SPFuzz and SPFuzz++ remain effective for computation-intensive applications such as `gzip`, `bsdtar`, `omnetpp`, etc. Upon closer examination, we have discovered that even though computation-intensive applications may not frequently use IO-related syscalls like *read* and *write*, they still rely on memory-related system calls such as *brk*. For instance, in `gzip`, we found that 77% of its *if* statements contain system calls in at least one branch. This indicates that system calls can determine the outcomes of 77% of branches in `gzip`. As a result, SPFuzz and SPFuzz++ can still successfully fuzz computation-intensive applications. Moreover, we also evaluated the performance of AFL-Dumb, a totally black-box fuzzer. As the data in Table 8 (in our GitHub repo) shows, SPFuzz outperforms AFL-Dumb on all benchmarks with at most 45% branch coverage advantage, demonstrating that the SPC feedback is effective. The conclusion for SPFuzz++ and AFL++-Dumb is the same since they only changed the fuzzer base, which mainly consists of the mutation and selection algorithms and does not affect this examination for the upper feedback mechanism.

To evaluate the performance of SPFuzz and SPFuzz++ in the worst case, we crafted a challenging target that only takes a file with a fixed length and calculates hashes with different hashing algorithms according to the content. We carefully removed the lines with syscalls from the branch statements, then SPFuzz and SPFuzz++did find no new patterns during the fuzzing process. We need to acknowledge that such specified targets are more suitable for traditional fuzzers. We will leave these extreme targets to future work.

7.3 External Libraries

An issue that could potentially undermine our approach is the fact that system calls are frequently encapsulated within libraries like `glibc`. Consequently, the system call sequences may not accurately represent the code executed by a program but rather the code executed within external libraries. Additionally, libraries may possess intricate internal logic, which could result in unstable system call patterns during fuzzing. For instance, different system call paths may be generated for the same input.

According to our evaluation, system call patterns are effective and stable approximations for code coverage in real-world applications that rely heavily on external libraries. The effectiveness of system call coverage is not compromised by the use of external libraries because we can determine which external library function is being called and, thus, reconstruct the code path executed of the program based on system call sequences using symbolic execution [45]. Although our approach does not explicitly recover the executed code path, the changes in the system call patterns are still sufficient to reflect the changes in executed code paths. Additionally, external libraries do not compromise the stability of fuzzing because most library functions are deterministic when the system environment remains constant. Even if these functions are wrapped in library calls, we can still obtain stable system call sequences.

7.4 SPC Collisions

Compared to traditional branch/edge-based code paths, syscall patterns have more collisions since several different syscall patterns may refer to one identical code path. Therefore, we recorded the number of syscall patterns from SPFuzz++ and traditional code paths from AFL++QEMU (which should be theoretically sound) when both reached the same branch coverage. This coverage was defined as the minimum coverage achieved by either after 24 hours of fuzzing, ensuring that corresponding data was available for comparison. As shown in Table 6, the numbers of syscall patterns are, on average, approximately twice as traditional AFL code paths. However, this does not suggest that SPFuzz and SPFuzz++ need a doubled efficiency to reach identical results. As concluded in Section 6.5, the distances between SPCs are positively correlated with the corresponding code coverage. Therefore, the distances between collided SPCs, i.e., SPCs representing identical code paths, are significantly smaller than those between the other SPCs and the collided ones. This leads to a lower priority for seeds whose SPCs collide with those already in the queue. Consequently, during the execution stage, seeds with higher priority for handling and mutation by SPFuzz and SPFuzz++ typically exhibit a lower collision rate.

In most situations, SPC collisions with significant distances (which is rare according to Section 6.5) undermine the fuzzing efficiency since the instrumentation-based feedback is more accurate. However, things can be different occasionally, where the SPC feedback captures details that escape from the observation of instrumentation-based feedback. For example, as shown in Figure 2 (b), SPFuzz++ gained a drastic coverage increase after about three hours since it prioritized a seed with frequent network-related syscalls but no new coverage and reached a key path through the mutations from this seed. This did not happen for AFL++QEMU even after accounting for the difference in execution speeds because the instrumentation-based feedback deprioritized the seed since it brought no new coverage and was bigger.

7.5 Specific Syscalls

Among the 300+ system calls available in Linux, some specific syscalls may bring uncertainty or meaningless information to the syscall sequence. Researchers of kernel fuzzing found that if a system call is invoked through *ioctl*, it becomes challenging to distinguish the sequence of system calls [42]. However, unlike kernel fuzzing, we focus more on real-world targets, where sequences containing frequent

Table 6: The average number of syscall patterns (from SPFuzz++) and the average number of traditional code paths (from AFL++QEMU) under the same branch coverage.

Benchmark	SPC Patterns	AFL Paths	Benchmark	SPC Patterns	AFL Paths
nginx	5669	4195	lua	9452	4918
gzip	1548	483	pdftotext	9342	5241
z3	10842	4440	tcpdump	15239	15529
readelf	5731	1800	bsdtar	190	121
cert-basic	680	103	sfconvert	469	389
unrtf	3389	1363	jasper	24	22
clean_text	189	105	perlbench_r	28800	11034
mcf_r	569	484	omnetpp_r	5123	2102
cpuxalan_r	1320	984	x264_r	442	109

identical syscalls(such as the aforementioned *ioctl*) are rare. Moreover, according to Section 4, unstable parameters will be pruned before the fuzzing campaigns start. Therefore, SPFuzz and SPFuzz++ can distinguish *ioctl* calls from remaining parameters such as cmd. This also applies to other similar syscalls.

7.6 Cross-Platform and Cross-Language Fuzzing

Though the main implementation and evaluations are done mainly on the Linux OS, system call pattern coverage is general to all OSes. Besides, although our implementation of SPFuzz and SPFuzz++ are for C/C++ programs, it makes no difference for SPFuzz and SPFuzz++to support other languages that generate native binaries, such as Go and Rust.

For languages that use runtimes, such as Java and Python, system call patterns may not be as effective as in native binaries because the runtime itself may add randomness in calling system calls. However, we can apply a similar idea of system call patterns to the runtime API calls. For example, we can use the Java API call patterns to fuzz Java binaries. We will leave the idea of fuzzing runtime-based languages in our future work.

7.7 Debugger-Based Method

Linux Strace [3], which is used by debuggers, is another approach for gathering code coverage without code instrumentation and not disrupting the program. However, researchers found that such a method suffers from overhead as high as 19 times [34] since it is based on the *ptrace* function of Linux [28]. Since *ptrace* uses system interruptions to get the runtime information of the program, its overhead can not be easily reduced [2].

8 Related Work

Fuzzing has become a widely adopted automatic software testing method [7–9, 35, 50, 67]. The most popular method of fuzzing currently is the instrumentation-based grey-box fuzzing [65]. However, this method requires access to source code for instrumentation and is therefore not directly available for Commercial Off-The-Shelf (COTS) programs. To address this limitation, researchers have proposed binary-only fuzzing techniques [4, 8, 21, 30, 39, 40, 58, 65]. However, these techniques all suffer from the limitations we mentioned in Section 2.

Auditing frameworks, such as LTTng [29], Retrofitting [17], and Trustworthy [20], have been widely used in various security tasks. These tasks include APT attack investigation [33, 62] and stealthy behavior detection [18, 64]. Most current frameworks are not effective for fuzzing since they will randomly drop system calls when the system is busy. Thereby, we adopt the latest framework, NoDrop, that ensures the integrity of syscall collection [41]. System calls can also be used by developers to gather debugging information [3], maintainers to monitor system status [31], and security experts for forensics on malicious activities or to reconstruct the cybercrime scenario after an attack [18–20]. These approaches inspired our work but can not be directly applied to binary-only fuzzing.

9 Conclusion

This paper introduces a novel approach called system call pattern coverage, which offers an alternative method for providing feedback in greybox fuzzing. This approach is particularly useful in situations where code coverage is not available, such as when dealing with closed-source binaries that are protected. Compared to existing feedback approaches, system call pattern coverage does not require static rewriting, emulation, or hardware support. Therefore, system call pattern coverage does not face the risks of breaking the logic of target applications, having too slow fuzzing speed, or introducing dependency on specific hardware. To validate the effectiveness of this feedback mechanism, we developed binary-only fuzzers called SPFuzz and SPFuzz++, which are capable of fuzzing a more significant number of (COTS) binaries compared to existing approaches. We conducted an evaluation of SPFuzz and SPFuzz++ using 29 real-world applications and found that they achieve levels of fuzzing effectiveness similar to those of the traditional code coverage guidance. In particular, during our evaluation, SPFuzz successfully identified six previously unknown Common Vulnerabilities and Exposures (CVEs), thus demonstrating its practical bug-hunting capabilities.

Acknowledgments

We want to thank the anonymous reviewers for their valuable feedback. Ding Li and Peng Jiang are the corresponding authors. This work was partly supported by the National Science and Technology Major Project of China (2022ZD0119103), the National Natural Science Foundation of China (62172009), and the CCF-Huawei Populus Euphratica Innovation Research Funding.

References

[1] 2014. URL: https://argp.github.io/2014/12/29/fuzzy-stack-hash/,title={Papernotes: Fuzzystackhash},.

[2] Linux ptrace syscall, 2018. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html.

[3] Linux strace, 2018. URL: https://github.com/strace/strace.

[4] More about afl: Binary-only instrumentation, 2019. URL: https://afl-1.readthedocs.io/en/latest/about_afl.html.

[5] "why is it so slow?" – ptfuzzer, 2019. URL: https://github.com/hunter-ht-2018/ptfuzzer/issues/12.

[6] Afl implementation, 2020. URL: https://github.com/google/AFL.

[7] Fuzz testing in chromium, 2022. URL: https://chromium.googlesource.com/chromium/src.git/.

[8] Honggfuzz, 2022. URL: https://honggfuzz.dev/.

[9] libfuzzer – a library for coverage-guided fuzz testing., 2022. URL: https://llvm.org/docs/LibFuzzer.html.

[10] AFL FAQ at Stability. https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/FAQ.md, 2023.

[11] Arm-based cpus could double notebook pc market share by 2027: Report., 2023. URL: https://www.tomshardware.com/news/arm-based-cpus-set-to-double-notebook-pc-market-share-by-2027.

[12] Afl++ qemu mode, 2024. URL: https://github.com/AFLplusplus/AFLplusplus/tree/stable/qemu_mode.

[13] Intel blsi cf computation bug, 2024. URL: https://gitlab.com/qemu-project/qemu/-/issues/2175.

[14] Perf wiki on intel pt supports, 2024. URL: https://perf.wiki.kernel.org/index.php/Perf_tools_support_for_Intel%C2%AE_Processor_Trace.

[15] Hervé Abdi. The kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pages 508–510, 2007.

[16] Haldun Akoglu. User's guide to correlation coefficients. *Turkish journal of emergency medicine*, 18(3):91–93, 2018.

[17] Adam Bates, Kevin Butler, Alin Dobra, Brad Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Retrofitting applications with provenance-based security monitoring. *arXiv preprint arXiv:1609.00266*, 2016.

[18] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *Proceedings of the 26th International Conference on World Wide Web*, pages 887–895, 2017.

[19] Adam Bates, Ben Mood, Masoud Valafar, and Kevin Butler. Towards secure provenance-based access control in cloud environments. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 277–284. ACM, 2013.

[20] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy {Whole-System} provenance for the linux kernel. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 319–334, 2015.

[21] Battelle., 2017. URL: https://github.com/Battelle/afl-unicorn.

[22] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[23] Andrea Biondo. Improving afl's qemu mode performance, 2018. URL: https://abiondo.me/2018/09/21/improving-afl-qemu-mode/.

[24] Gianluca Borello. System and application monitoring and troubleshooting with sysdig. Washington, D.C., November 2015. USENIX Association.

[25] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 633–645, 2019.

[26] Codenomicon. Heartbleed bug, 2014. URL: https://heartbleed.com/.

[27] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 53–63, 2006.

[28] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU pitfalls: Attacks on PKU-based

memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426. USENIX Association, August 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/connor.

[29] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224. Citeseer, 2006.

[30] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, 2020. doi:10.1109/SP40000.2020.00009.

[31] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1285–1298, 2017.

[32] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT'20, USA, 2020. USENIX Association.

[33] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. SAQL: A stream-based query system for real-time abnormal system behavior detection. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 639–656, Baltimore, MD, August 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/gao-peng.

[34] Mohamad Gebai and Michel R Dagenais. Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Computing Surveys (CSUR)*, 51(2):1–33, 2018.

[35] Google. syzkaller - kernel fuzzer, 2016. URL: https://github.com/google/syzkaller.

[36] William H Hawkins, Jason D Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W Davidson. Zipr: Efficient static binary rewriting for security. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 559–566. IEEE, 2017.

[37] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020. doi:10.1145/3428334.

[38] Ningyu He, Zhehao Zhao, Jikai Wang, Yubin Hu, Shengjian Guo, Haoyu Wang, Guangtai Liang, Ding Li, Xiangqun Chen, and Yao Guo. Eunomia: Enabling user-specified fine-grained search in symbolically executing webassembly binaries. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 385–397, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3597926.3598064.

[39] Marc Heuse. Afl-dynamorio, 2018. URL: https://github.com/vanhauser-thc/afl-dynamo.

[40] Marc Heuse. Afl-pin, 2018. URL: https://github.com/vanhauser-thc/afl-pin.

[41] Peng Jiang, Ruizhe Huang, Ding Li, Yao Guo, Xiangqun Chen, Jianhai Luan, Yuxin Ren, and Xinwei Hu. Auditing frameworks need resource isolation: A systematic study on the super producer threat to system auditing and its mitigation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 355–372, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-peng.

[42] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[43] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[44] Ankur Limaye and Tosiron Adegbija. A workload characterization of the spec cpu2017 benchmark suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 149–158. IEEE, 2018.

[45] Xuanzhe Liu, Chengxu Yang, Ding Li, Yuhan Zhou, Shaofei Li, Jiali Chen, and Zhenpeng Chen. Adonis: Practical and efficient control flow recovery through os-level traces. *ACM Trans. Softw. Eng. Methodol.*, jul 2023. Just Accepted. doi:10.1145/3607187.

[46] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[47] METABASE. Imagemagick cves, 2016. URL: https://www.metabaseq.com/imagemagick-zero-days/.

[48] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for ebpf-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151, 2021.

[49] Michael Larabel. phoronix-test-suite. https://github.com/phoronix-test-suite/phoronix-test-suite, 2024.

[50] Micheal Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl, 2017.

[51] Bruno Morisson. Analysis of the linux audit system. *Master's thesis, Information Security Group, Royal Holloway, University of London*, 2014.

[52] Stefan Nagy. Zax failure on "zola" binary., 2021. URL: https://git.zephyr-software.com/opensrc/zafl/-/issues/34.

[53] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

[54] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium*, 2021.

[55] Nikolaj Bjorner. z3Test. https://github.com/Z3Prover/z3test, 2024.

[56] Philip Sedgwick. Pearson's correlation coefficient. *Bmj*, 345, 2012.

[57] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. Vancouver, BC, August 2017. USENIX Association.

[58] Talos-vulndev. Afl-pin, 2018. URL: https://github.com/talos-vulndev/afl-dyninst.

[59] AFL++ Team. Notes for asan., 2023. URL: https://aflplus.plus/docs/notes_for_asan/.

[60] AFL++ Team. Afl++ nyx mode, 2024. URL: https://github.com/AFLplusplus/AFLplusplus/blob/stable/nyx_mode/README.md.

[61] Joshi Vaibhav Vijay and Balbhim Bansode. Arm processor architecture. *International Journal of Science, Engineering and Technology Research (IJSTR)*, 4(10), 2015.

[62] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A Gunter, et al. You are what you do: Hunting stealthy malware via data provenance analysis. In *NDSS*, 2020.

[63] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[64] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications. In *NDSS*, 2020.

[65] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book, 2019.

[66] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 6:37302–37313, 2018.

[67] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 659–676, 2021. doi:10.1109/SP40001.2021.00109.

[68] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 659–676. IEEE, 2021.

[69] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.