Configuration-Sensitive Linux Kernel Fuzzing

Yuheng Shen
KLISS, BNRist, School of Software,
Tsinghua University
Beijing, China
syh1308@gmail.com

Jianzhong Liu
School of Computer Science,
Shandong University
Qingdao, China
liujianzhong@sdu.edu.cn

Yuhan Chen
School of Electronic Information,
Central South University
Changsha, China
Chenyuhan@csu.edu.cn

Yifei Chu
KLISS, BNRist, School of Software,
Tsinghua University
Beijing, China
chuyf24@mails.tsinghua.edu.cn

Qiang Zhang
College of Computer Science,
Hunan University
Changsha, China
zhangqiang9413@126.com

Guoyu Yin
School of Electronic Information,
Central SouthHunan University
Changsha, China
yinguoyu@csu.edu.cn

Heyuan Shi
School of Electronic Information,
Central South University
Changsha, China
hey.shi@foxmail.com

Yu Jiang
KLISS, BNRist, School of Software,
Tsinghua University
Beijing, China
jiangyu198964@126.com

Abstract

Fuzzing operating system kernels to discover deep and complex bugs is difficult to accomplish, as kernels reside between hardware and user applications, exposing a variety of input vectors that affect their internal state. Previous approaches mainly use a kernel's system call interface to deliver test payloads into the kernel, but reaching states and triggering bugs also require collaborative efforts from other input vectors, such as runtime parameters. Kernel runtime parameters greatly affect the execution behavior of the kernel under test, as they alter internal execution flows and thus require kernel fuzzers to manipulate them in conjunction with invoking system calls to effectively root out bugs. In this paper, we present CSGO, a kernel fuzzer that discovers more in-depth bugs through fuzzing kernel runtime parameters in conjunction with system calls. CSGO's approach is achieved through the following designs. First, CSGO generates valid test case generation syntax for kernel configurations by extracting available parameters exposed by the kernel. Then, for the extracted parameters, CSGO statically deduces relations between the configurations and kernel system calls for initial generation guidance. Finally, during fuzzing, CSGO dynamically refines the relations between the configurations and system calls by interpreting execution feedback. We implemented CSGO and evaluated its approach on recent versions of the Linux kernel. Our results show that CSGO achieves an average of 21% improvement in overall coverage compared with existing state-of-the-art kernel fuzzers and triggers 22 previously unknown bugs, with 8 fixed by kernel maintainers.

Heyuan Shi and Yu Jiang are the corresponding authors.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
ICSE '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773136

Keywords

Fuzz Testing, Linux Kernel, Bug Detection

ACM Reference Format:
Yuheng Shen, Jianzhong Liu, Yuhan Chen, Yifei Chu, Qiang Zhang, Guoyu Yin, Heyuan Shi, and Yu Jiang. 2026. Configuration-Sensitive Linux Kernel Fuzzing. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3773136

1 Introduction

The Linux kernel is important in modern infrastructure. Linux is widely utilized across a range of platforms, from servers to embedded devices, underscoring its importance within the software stack. Given its critical role, the correctness and reliability of the kernel are paramount, as any defects could lead to severe consequences, such as system failures and potentially life-threatening situations. However, the extensive and complex nature of the kernel codebase presents significant challenges to maintaining a bug-free system. For instance, Linux 6.13 includes over 27.9 million lines of code, making it difficult to review and check the code manually.

Fuzz testing, also known as fuzzing [3, 16, 21], is an automated testing technique for finding bugs. Over the past decade, building an effective kernel fuzzer has been a topic of interest in the research community. Many existing fuzzers [20, 29] have been incorporated into the kernel's CI/CD pipeline to ensure the kernel's quality. Current kernel fuzzers usually use system call descriptions, also known as Syzlang [30], as test input, by choosing random system calls and their arguments to generate test cases. Then, fuzzers use code coverage to determine whether the test case triggers new code paths in the kernel and give those inputs higher priority for further testing. State-of-the-art kernel fuzzers such as Syzkaller [29] and Moonshine [20] have found many bugs in the kernel and have been widely used in the kernel community.

While current fuzzers are effective in identifying many vulnerabilities in the kernel, their focus is mostly on system call-based

methods. However, this pure system call-based fuzzing approach restricts their testing to a kernel's dynamic configurations, limiting their testing capabilities to a confined attack surface. Consequently, these fuzzers may fail to detect bugs that are deeply embedded within the kernel's logic, manifesting only under specific and sophisticated runtime conditions. For instance, certain kernel bugs may only appear when specific dynamic configurations (runtime-adjustable settings exposed via `sysfs` or `/proc/sys` that change subsystem behavior without reboot) are set to unusual values or when particular subsystems are activated. Under normal circumstances, these conditions might not be met during routine fuzzing sessions, allowing potential vulnerabilities to remain undetected. As a result, the kernel contains latent bugs that emerge only under certain runtime conditions.

Specifically, apart from system calls, the kernel offers an extensive range of dynamic configurations. In our initial analysis, we discovered around 2,300+ dynamic configurations for Linux 6.1 that can affect the execution of the kernel. These configurations encompass functionalities from network-related settings (e.g., TCP settings) and security directives (e.g., toggles for SELinux or AppArmor) to scheduler configurations and device-specific attributes. Even simple modifications to a single parameter can alter the kernel's execution paths, thereby changing the overall kernel state space. Take the dynamic configuration `tcp_syncookies` as an example, as shown in Figure 1. Under the same system workload, by default, the execution path is A to C. However, when we change the parameter value to 2, the execution path changes from A to B to C. Therefore, toggling this configuration can change the kernel's behavior. Despite the potential impact of these thousands of configurations, they are often overlooked or entirely ignored in typical fuzzing campaigns. By not adjusting for how these configurations can transform kernel behavior, fuzz testing tools miss valuable opportunities to detect bugs in the kernel under a diverse set of real-world conditions. Therefore, incorporating these dynamic configurations into the fuzzing workflow can help uncover hidden issues that purely system call-driven testing may never encounter.

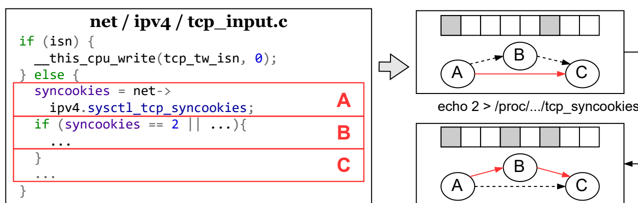

Figure 1: A Code example shows the impact of configuration on the execution path. When `tcp_syncookies` is set to 2, the execution path changes from `A->C` to `A->B->C`.

In the kernel domain, there are two types of configuration: static and dynamic configurations. Static configurations are set via Kconfig at compile time and determine whether entire kernel subsystems or features are included, they are fixed post-boot. In contrast, dynamic configurations are mutable at runtime and influence kernel behavior based on current system state. As such, static and dynamic configurations expose orthogonal regions of the kernel state space, the former controls structural presence, while the latter shapes runtime execution. Prior works have mainly addressed configuration-related bugs from a static configuration perspective. For example, Configfix [10] resolves compile-time conflicts using SAT-based reasoning, while Abal [1] found that most such misconfigurations can lead to kernel errors. For dynamic configuration, recent works like KeenTune [32] use dynamic configurations for performance tuning, they do not explore fault-triggering behaviors. Our work aims to test the Linux's dynamic configurations alongside system calls, uncovering bugs that depend on runtime-specific conditions unreachable by static configuration fuzzers.

To effectively leverage the kernel's dynamic configuration to assist fuzzers in rooting out bugs, we need to address the following challenges: **1. Generate valid configuration payloads.** The first challenge is to generate valid dynamic configuration payloads. In detail, dynamic configurations are often exposed via files within the virtual file system, such as `/proc/sys` and `/sys`. However, such files are scattered across these directories and are not well-organized or documented, making them difficult to identify and extract. Furthermore, once extracted, it is challenging to identify the values for each configuration, as these values are often diverse with different types, such as integer and string. However, the configuration files are often not well-formatted, and related documentation is lacking, which complicates the identification and understanding of these values and settings. To generate valid configuration payloads, we need to extract the configuration parameters from the kernel and understand the valid range of values for them.

2. Identify the relation between configuration and system calls. Once we have the valid configuration payloads, the next challenge is to understand the relation between configurations and the system calls. In detail, the kernel has thousands of configurations and system calls, i.e., Linux 6.1 has around 2,300+ dynamic configurations, and Syzkaller has 1000+ system call specifications. Randomly toggling configurations alongside arbitrary system call sequences often leads to redundant and nonsensical combinations, wasting computing resources and failing to expose deeper kernel bugs. For instance, tweaking a TCP congestion control setting migh t have no observable impact on a purely filesystem-oriented system call sequence. Without a structured method to pinpoint or learn these configuration–system call relationships, fuzzing campaigns can have poor efficiency.

In order to address the above challenges, we propose CSGO, a configuration-sensitive kernel fuzzer that leverages dynamic kernel configurations to improve fuzzing effectiveness. In detail, before the fuzzing process, CSGO first extracts the dynamic configurations from the kernel, and based on each extracted configuration, CSGO synthesizes the corresponding configuration calls, which are special pseudo-system calls designed to modify kernel parameters at runtime. Then, CSGO analyzes the associations between configuration calls and system calls using static mapping deduction and generates a relevance weighting table. During fuzzing, based on the weighting table, CSGO generates test payloads that combine system calls and configuration calls and feeds payloads to the kernel. By using the dynamic mapping detection at runtime, CSGO can further identify the runtime relation between configuration calls and system calls, and CSGO can update the weighting

table and generate more effective test payloads. This helps CSGO to explore the kernel's state space more effectively.

For evaluation, we demonstrate the viability of CSGO by evaluating the bug detection capability and achieved code coverage, and by comparing it to three state-of-the-art kernel fuzzers on four Linux versions, i.e., Linux 6.13, 6.12, 6.10, and 6.8. In summary, CSGO detected 22 previously unknown bugs in the Linux kernel, with 10 of which confirmed, 8 fixed by the kernel maintainers, and 2 are assigned with CVE IDs [18, 19]. In addition, compared with vanilla Syzkaller and Moonshine, CSGO improves coverage by 22%, 33%, and 9% for Syzkaller, Moonshine, and Actor [9], respectively.

In summary, the contributions of this paper are as follows:

- We propose a new approach that leverages the kernel's dynamic configuration during fuzzing. This allows us to test the kernel under a diverse set of real-world conditions and uncover hidden bugs.
- We propose CSGO, a configuration-sensitive kernel fuzzing framework that leverages dynamic kernel configurations and configuration-system calls relevance to enhance the overall fuzzing performance.
- The results show that CSGO can detect 22 previously unknown bugs in the Linux kernel, with 8 of them being fixed by the kernel maintainers.

2 Background

In this section, we introduce the concept of dynamic configurations in Linux, and provide a brief overview of kernel fuzzing.

2.1 Dynamic Configurations in Linux

The Linux kernel provides extensive support for dynamic configurations that can be modified at runtime via virtual file systems. These dynamic configurations, accessible primarily through interfaces like `sysfs` and `procfs`, expose tunable parameters across diverse subsystems, including networking, memory management, security frameworks, and device-specific configurations. Unlike static configurations set at compile time, dynamic configurations allow real-time adjustments without requiring a system reboot, granting system administrators and automated management tools significant flexibility to optimize performance, adjust to varying workloads, and strengthen security measures dynamically.

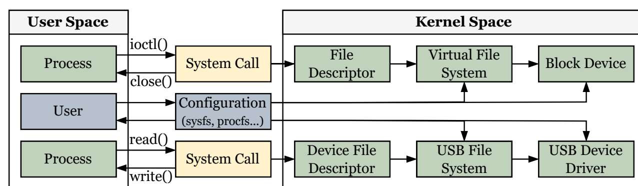

Figure 2: Illustration of Linux's dynamic configuration.

Dynamic configurations serve as an important interface within the kernel, bridging user-space processes and kernel internals, as illustrated in Figure 2. Although similar to system calls in providing an interface for user-kernel interaction, dynamic configurations differ fundamentally: system calls invoke specific kernel functions, while dynamic configurations modify kernel policies, parameters, and internal settings that govern system-wide behavior. By altering these configurations, users gain the ability to finely tune kernel operations to address specific requirements, resolve performance issues, or trigger kernel behaviors unattainable by system calls alone. For instance, adjusting memory management settings by writing to `/proc/sys/vm/swappiness` directly influences swap usage patterns, improving memory efficiency under different workload scenarios. Similarly, modifying device-specific settings such as USB power management parameters through `/sys/module/usbcore/parameters/autosuspend` helps optimize power consumption and troubleshoot connectivity issues dynamically.

2.2 Kernel Fuzzing

Fuzz testing, also known as fuzzing, is an automatic program testing technique that provides random data as input to the target program and monitors the program's behavior, for effects such as crashes, hangs, or memory leaks, to detect potential bugs. Fuzzing has been widely used to detect bugs in software [2, 8, 16, 24, 37], and has been proven to be effective in finding bugs in the Linux kernel. Traditional kernel fuzzing [5, 14, 23, 26, 27] usually uses system calls as the input vector, then they use code coverage to determine whether the test case triggers new code paths in the kernel, and give those inputs higher priority for further testing. Currently, the most famous kernel fuzzer, Syzkaller, has been integrated into kernel's CI/CD pipeline to ensure the kernel's quality.

In detail, Syzkaller leverages system call descriptions, known as Syzlang [30], to define how test payloads should be generated. Each Syzlang file contains necessary header files, required system resources, structures, and a set of system calls. Based on the Syzlang, Syzkaller generates test payloads that simulate the kernel's real-world execution. During fuzzing, Syzkaller sends the generated test payloads to the target kernel. The executor running within the target kernel then interprets the test payloads, invokes the target system call using function pointers, and monitors the kernel's execution. After execution, Syzkaller collects feedback information, such as code coverage and detected bugs. Based on this feedback, Syzkaller saves those test payloads that trigger new system behaviors, assigning them a higher weighted value for further mutation and execution, and discards those test payloads that do not trigger new system behaviors.

To simulate complex system behaviors that are hard to achieve through primitive system calls alone, Syzkaller supports pseudo system calls, which are a set of user-implemented functions that directly run within the executor. To use the pseudo system call, the user needs to provide the pseudo system call's implementation and the corresponding Syzlang description to Syzkaller, and Syzkaller will use the provided information to generate test payloads that can trigger the pseudo system call during fuzzing.

3 Motivation

In this work, we focus on leveraging the Linux kernel's dynamic configurations to synthesize payloads that enable a more effective exploration of the kernel's state space. However, the Linux kernel contains thousands of dynamic configurations, where randomly selecting a configuration and a system call could lead to poor testing performance. For a more concrete statistic, the Linux

6.13 provides 450 system calls, and Syzkaller, at the time of writing, defines 4,499 corresponding system call implementations. In addition, there are at least 3,565 dynamic configurations available in the `/proc/sys` and `/sys` directories. Consider a typical system call sequence usually consisting of 10 to 32 individual calls, the total number of possible configurations and system call combinations is around $\sum_{L=10}^{32} \left(N_{\text{syscall}} \times N_{\text{config}}\right)^{L}$, the total number would be around 10^{232}, where N_{syscall} is the total number of system call, and N_{config} is the total number of configurations, and L is the length of the system call sequence. Given that Syzkaller can execute around 100 system call sequences per second, it would take around 10^{223} years to explore the entire configuration and system call combination space. Such a huge search space makes it infeasible to explore all possible configurations and system call combinations.

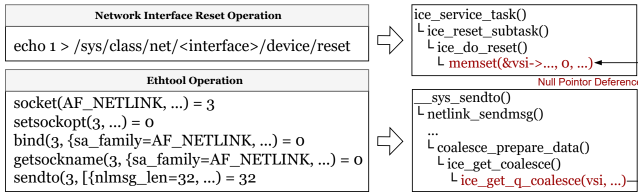

Figure 3: Illustration of CVE-2024-46770.

However, not all configuration and system call combinations are equally important , only under certain combinations of configurations and system calls would the kernel expose potential bugs; we use CVE-2024-46770 depicted at Figure 3 as a motivating example. In detail, this bug is located in the network module. To trigger the bug, the attacker first induces a state change in the network's dynamic configuration that triggers a reset operation. During this process, the network interface transitions into an intermediate state where some internal structures are partially reset, but the interface has not yet been globally detached from the network core. Then, during this resting period, if another socket operation is executed, it will access the interface's data that is being reset. Since the reset procedure has not been fully completed and the global state remains inconsistent, this access can lead to a null pointer dereference, triggering the bug. As we can see from the motivating example, to coordinate the configuration and system call, we need to (1) choose the proper configuration with a value that can cover the potential error state and (2) understand the relation between the configuration and the system call and generate effective test payloads that combine configurations and system calls. Concretely, to leverage the kernel's dynamic configuration in fuzzing, we need to address the following challenges:

Generate Valid Configuration Payloads. To effectively utilize dynamic configurations in kernel fuzzing, the first challenge is to accurately extract and generate valid dynamic configuration payloads that can induce specific kernel behaviors. Dynamic configurations in the Linux kernel are typically exposed via files within virtual filesystems such as `/proc/sys` and `/sys`. However, identifying which files contain configuration information is inherently difficult because these files are scattered throughout various directories and often lack clear, structured documentation or consistent formatting. Furthermore, configurations may possess diverse value types, such as integers, booleans, and strings, and each type can have distinct semantics and constraints. Without precise knowledge of the valid value ranges and expected formats for each configuration, fuzzers risk generating invalid payloads that fail to trigger meaningful or interesting kernel behaviors. As we can see from the motivating example, to trigger the bug, the user first needs to identify the corresponding network configuration that can trigger the reset operation and have the corresponding value that can induce the error state. Therefore, we need to accurately identify and extract this dynamic configuration in the kernel to generate a valid configuration payload.

Perceive the Relevance Between Configurations and System Calls. After generating valid configuration payloads, the next challenge is to understand the relevance between dynamic configurations and system calls to construct meaningful test inputs. The Linux kernel exposes thousands of configuration parameters and hundreds of system calls, resulting in a vast and sparse combination space. As shown in our motivating example, certain bugs only surface when specific configurations interact with system calls in precise sequences. Randomly combining them is inefficient and often ineffective. While static analysis could theoretically infer these dependencies, it is largely impractical in practice. Even when limited to known registration patterns, it must resolve pointer aliasing, macro expansions, and deep struct nesting, often spread across translation units and obscured by preprocessor logic. Additionally, many configurations affect control flow only under specific runtime conditions, such as memory pressure or socket state, which static analysis cannot capture. Therefore, dynamic inference is essential for mapping configuration and system call interactions and enabling targeted fuzzing that reaches deeper kernel states.

4 Design

In this paper, we propose CSGO, a configuration-sensitive kernel fuzzing framework that leverages dynamic kernel configurations to enhance the overall fuzzing performance.

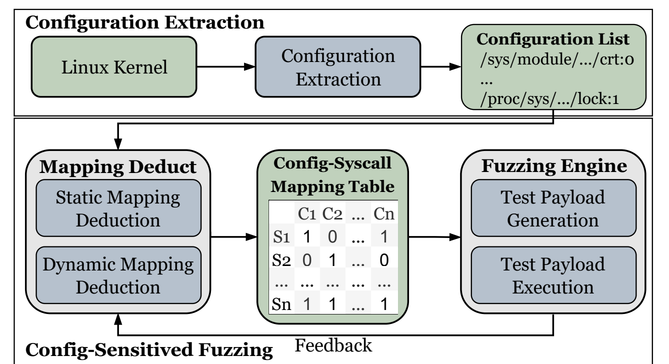

Figure 4: Workflow of CSGO.

The overall workflow of CSGO is shown in Figure 4, and consists of two phases: dynamic configuration extraction and configuration-sensitive fuzzing. In the extraction phase, CSGO performs a dry run of the target kernel to scan system interfaces and collect a list of writable configuration files and their values. From this data,

CSGO generates a configuration corpus composed of user-defined pseudo-functions, each capable of setting a specific configuration value. It also constructs an initial configuration–system call weighting table that captures potential relevance between configuration entries and system calls. In the fuzzing phase, CSGO uses this table to guide test generation, constructing payloads that combine relevant configuration writes with system calls. During each fuzzing iteration, CSGO gathers feedback, such as code coverage and crash reports, and updates the weighting table based on observed execution behavior, enabling iterative refinement of configuration-system call relations and deeper exploration of kernel states.

4.1 Configuration Extraction

The goal of configuration extraction in CSGO is to identify dynamic kernel configurations exposed via the virtual file system and synthesize corresponding call descriptions that can be used during fuzzing. To achieve this, CSGO performs a dry run of the target kernel to enumerate accessible configuration files under directories like `/proc/sys` and `/sys`. Each extracted configuration is represented as a tuple ⟨`M`, `P`⟩, where `M` denotes the associated kernel module, and `P` is a vector of ⟨`Arg`, `Type`⟩ pairs that describe the argument names and their types. Similarly, each system call is represented as a tuple ⟨`M`, `D`⟩, where `M` again refers to the kernel module, and `D` is a vector of ⟨`Name`, `Arg`, `Type`⟩ describing the system call name and its arguments. These unified representations allow CSGO to later analyze and match configuration entries with system calls for relevance inference and guided fuzzing.

Dynamic Configuration Extraction. To extract dynamic configurations from the kernel, CSGO scans the `/sys` and `/proc/sys` directories to identify configuration files that can be modified at runtime. This scanning process evaluates each file's writability, sinc e those read-only files are unlikely to be changed and are less relevant for fuzzing. Next, CSGO examines the content of each file to determine its data format. Given the fact that configuration values can vary widely, CSGO restricts extraction to numeric variables because such configurations have a well-defined, finite range that can be manipulated during fuzzing, whereas string variables are harder to generate reliably and lack formal documentation of valid options. Files meeting these criteria are then added to the configuration list, with CSGO recording the corresponding file path, configuration name, type, and valid range of values.

In detail, CSGO first initializes an empty configuration list and iterates over the `/sys` and `/proc/sys` directories to locate configuration files. For each file, it verifies writability and reads the file's contents. If the content represents an integer, CSGO classifies the configuration as an integer type and adds it to the list. In cases where the content is formatted as a key-value pair (e.g., `PCI_CLASS=10180`), CSGO records it as a key-value configuration. Finally, the completed configuration list is returned, providing the dynamic configurations that can be used in the fuzzing process.

Configuration Call Synthesis. Once CSGO acquires the configuration list, it generates configuration call descriptions and corresponding pseudo functions that can be used in the fuzzing process. Specifically, for each configuration in the list, CSGO extracts the configuration's name and type and generates a configuration call that sets the configuration to a random value and a reset call that restores the configuration to its default value. CSGO then generates a configuration call description that indicates the configuration's name and type. The configuration call description is stored in the initial specification list, and the configuration call and reset call are added to the configuration corpus.

4.2 Configuration Relevance Deduction

To minimize the search space and improve the efficiency of the fuzzing process, CSGO needs to perceive the relevance between configuration and system call so that it can generate an effective test payload that combines configuration and system call. To this end, CSGO first conducts a static mapping deduction, by analyzing the targeting module of the configuration and the system call, CSGO generates a initial weighting table that indicates the potential relevance between configuration and system call. Furthermore, during fuzzing, based on the feedback information such as code coverage, CSGO conducts dynamic mapping deduction to unveil the relation between system calls and configurations and updates the table to guide further generation.

Static Mapping Deduction. The goal of static mapping deduction is to generate an initial weighting table that indicates the potential relevance between configurations and system calls. If a configuration and a system call targets the same module, CSGO marks them as relevant. Concretely, considering $s = \langle M(s), D(s) \rangle$ and $c = \langle M(c), P(c) \rangle$, the relevance score is as follows:

$$R(s,c) = \frac{\mathrm{LCS}(M(s), M(c))}{\max\{|M(s)|,\ |M(c)|\}} \times \frac{v(M(s)) \cdot v(M(c))}{\|v(M(s))\| \cdot \|v(M(c))\|} \quad (1)$$

This composite score `R(s,c)` lies in the interval `[0,1]` and encapsulates both the structural and semantic similarities between the module names. The first term computes the normalized longest common substring (LCS) similarity, reflecting the degree of character level overlap, while the second term calculates the cosine similarity between the module names, capturing their semantic proximity. In detail, such metric identifies similarity between different texts regardless of their length, so it is robust to sizes (e.g., /sys/devices/virtual/ptp/ptp0/max_vclocks vs. openat$ptp0). For example, if the module names of a system call and a configuration share a significant common substring and exhibit high cosine similarity in their embeddings, `R(s,c)` will be close to 1, indicating a strong relevance. Here, CSGO deems a system call and a configuration as relevant if `R(s,c)` exceeds a predefined threshold of 0.6.

For the initial weighting table `MP`, CSGO assigns a relevance type to each system call and configuration pair based on the relevance score `R(s,c)`. In detail, if $R(s,c)$ exceeds the threshold τ, the relevance type is set to 15, indicating a strong relevance. If there exists another system called s' targeting the same module as s and sharing a strong relevance with c, $MP(s',c)$ is set to 10, indicating a moderate relevance. Otherwise, the relevance type is set to 0, indicating weak relevance. This process generates an initial weighting table that outlines the potential relationships between configurations and system calls, allowing CSGO to narrow down the search space and combine configurations and system calls effectively.

Dynamic Mapping Deduction. Once the fuzzing starts, CSGO conducts dynamic mapping deduction and updates the weighting

table based on the coverage statistics. In detail, based on the original execution payload, CSGO generates a mutated payload that combines the original payload with the corresponding configuration calls. Then, CSGO executes the mutated payload and collects the code coverage between the original payload and the mutated payload. Based on the coverage information, CSGO updates the weighting table and assigns a higher priority to the corresponding configuration call-system call elements that can increase the coverage and decreases the priority to the configuration calls that have not been observed to increase the coverage.

Algorithm 1: Coverage Guided Weighted Value Update

```
   Input: p1: Original payload sequence (e.g., [S1, S2, ..., Sn])
   Cov1: Coverage metric from executing p1
   Output: ct: Updated configuration choice table
1  Function UpdatePriorityTable(p1, Cov1):
2      Si ← p1[rand(len(p1))] ;   // select a system call
3      Ci, resetCi ← choiceConfigcall(Si) ; // select config
4      p2 ← insertConfigCall(p1, Si, Ci, resetCi);
5      Cov2 ← Exec(p2) ;          // get runtime coverage
6      if Cov1 < Cov2 then
           // get system call and configuration
              call's index
7          syscall_idx ← getSyscallIndex(Si);
8          config_idx ← getConfigcallIndex(Ci);
           // update the configuration choice table
9          MP[syscall_idx][config_idx] ←
            MP[syscall_idx][config_idx] + 5
10     else
           // lower priority, if coverage no changes
11         MP[syscall_idx][config_idx] ←
            MP[syscall_idx][config_idx] − 1
12     return MP;
```

The detailed process is illustrated in Algorithm 1. By randomly selecting a system call S_i from the original payload p_1 (line 2), then using a helper function to choose a corresponding configuration call C_i along with its reset call $resetC_i$ (line 3). Next, these calls are inserted into p_1 around S_i to form a mutated payload p_2 (line 4), which is then executed to obtain a new coverage metric Cov_2 (line 5). The algorithm compares Cov_1 (from p_1) with Cov_2; if the two differ (line 6), indicating that the configuration call has an impact on the system call, the weighting table entry corresponding to (S_i, C_i) is increased by 5 (line 9). Otherwise, if there is no difference in coverage, the weighting table entry is decreased by 1 (line 11). Finally, the updated weighting table is returned (line 12), guiding further payload generation by emphasizing configuration–system call pairs that meaningfully affect kernel behavior.

4.3 Configuration-Sensitive Generation

During fuzzing, CSGO needs to generate test payload that contain both system call and configuration and feeds it to the kernel.

Payload Generation. To generate test payloads that contain system call and configuration, CSGO first randomly selects a system call from the system call list, and then, based on the weighting table, CSGO chooses a configuration call that is relevant to the selected system call. Last, CSGO generates the payload that combines the selected system call and corresponding configuration call.

Algorithm 2: Weighted Configuration Call Selection

```
   Input: MP: The static weighting table
1  Function choiceConfigcall(syscall):
2      syscall_idx ← GetSysCallIndex(syscall);
3      row ← MP[syscall_index] ;   // get system call's
        location in the table
4      runningSum ← 0 ;    // compute cumulative value
5      for j ← 0 to row.len() − 1 do
6          runningSum ← runningSum + row[j];
7          row[j] ← runningSum;
8      runSum ← row[row.len() − 1];
9      if runSum > 0 then
10         x ← rand(runSum) + 1;
           // search the smallest index where the
              cumulative sum is at least x
11         res ← BinarySearch(row, x);
12         config_call ← GetConfigCall(res) ;        // get
            corresponding config call
13         return config;
14     else
15         return GetConfigCall(rand(len(MP[0])))
```

The detailed algorithm is shown in Algorithm 2. In particular, CSGO takes a system call as input and uses a static weighting table MP to select a corresponding configuration call. First, it retrieves the index of the given system call in the weighting table via `GetSysCallIndex` (line 2) and extracts the corresponding row from MP (line 3). Then, it computes a cumulative sum for that row by iterating through each element (lines 4–7), which effectively transforms the row into a cumulative distribution of weighted values. If the total cumulative sum (runSum) is greater than zero (line 9), a random integer x between 1 and runSum is generated (line 10). A binary search is then performed on the cumulative row (line 11) to find the smallest index at which the cumulative sum is at least x. The resulting index is mapped to the corresponding configuration call via `GetConfigCall` (line 12), and that configuration call is returned (line 13). Otherwise, if runSum is zero, a random configuration call is selected from the available options (line 15). This approach ensures that configuration calls are chosen in a weighted, randomized manner according to their assigned priorities.

4.4 Implementation

We implemented CSGO by extending Syzkaller, reusing core components such as the executor and corpus manager, and customizing key modules to support configuration-sensitive fuzzing. The system consists of two main components:

Configuration Extractor. We boot the target kernel in QEMU and use Python scripts to identify writable configuration entries in `/proc/sys` and `/sys`. Each entry is examined to extract permission and structural metadata. We then use regular expressions to translate the entries into pseudo-functions and Syzlang call descriptions, enabling configuration writes to be expressed as valid fuzzing operations. These pseudo-functions are inserted into Syzkaller’s system call interface, allowing dynamic configurations to be incorporated into fuzzing inputs like regular system calls.

Configuration-Sensitive Fuzzing. During the fuzzing process, CSGO builds a weighted relevance table that captures the likelihood of interaction between dynamic configurations and system calls. This table guides the generator to prioritize configuration calls that are more likely to affect the selected system call. CSGO augments Syzkaller’s default input construction by pairing system calls with relevant configuration writes to form combined payloads. After execution, a deflaking process compares coverage between the original and mutated inputs to reduce nondeterminism. Coverage differences are used to refine the weighted relevance table, allowing the fuzzer to iteratively improve its configuration–system call pairing strategy. This dynamic feedback loop enables CSGO to reach deep kernel states influenced by runtime configurations and uncover bugs beyond the scope of static configuration fuzzing.

5 Evaluation

In order to evaluate the effectiveness of CSGO, we conducted a series of experiments on recent versions of the Linux kernel. First, we examined CSGO’s bug detection capabilities by listing previously unknown bugs and presenting case studies on its findings. Second, we demonstrate CSGO capabilities to explore more execution paths and kernel state space by comparing the coverage information between CSGO and other existing tools. Last, we evaluate the contributions of each component to its ability to explore the kernel state space and the correctness of the identified relations. We design experiments for the following research questions:

- **RQ1:** How does CSGO perform in bug detection?
- **RQ2:** How does CSGO perform in exploring the kernel state space compared to other existing tools?
- **RQ3:** What is the effectiveness and accuracy of each component of CSGO.

5.1 Evaluation Setup

We conduct our evaluation of CSGO on four recent versions of the Linux kernel, namely Linux 6.8, 6.10, 6.12, and 6.13. We compare CSGO with three state-of-the-art kernel fuzzers, Syzkaller (`Commit ID 489e2d`) and Moonshine, and Actor, where Actor [9] is a recent kernel fuzzer that uses kernel’s memory allocation and free patterns to guide the test case generation.

To answer **RQ1**, all kernels are compiled with the default configuration, with `CONFIG_KCOV` [25] enabled to collect code coverage information and with `CONFIG_KASAN` [11] enabled to detect memory corruption bugs. To answer **RQ2**, we further implement CSGO on top of Moonshine (denoted as CSGO-ms) and compare the code coverage and number of unique bugs that CSGO and CSGO-ms achieved with vanilla Syzkaller, Moonshine and Actor. To answer **RQ3**, we first remove the mapping deduction component from CSGO (denoted as CSGO-NoMD) and CSGO-ms (denoted as CSGO-MsNoMD) and compare the code coverage and number of unique bugs detected between CSGO, CSGO-NoMD and CSGO-MsNoMD. We then sample and statistic the identified relation between configurations and system calls to evaluate the effectiveness of the mapping deduction component.

We perform the evaluation on a 64-core AMD EPYC 7742 CPU running Ubuntu 22.04 server. For bug detection, we run CSGO on the latest Linux kernel release version (6.13) for a week to detect previously unknown bugs. For the other comparisons, we run each fuzzer with the same QEMU configurations (2 VM, with 2 CPU for each fuzzing instance) for 24 hours, each repeated for five times, and following the fuzzing evaluation best practices [15].

5.2 Bug Detection Capability

To answer **RQ1** and evaluate CSGO’s bug detection capabilities, we collected and analyzed the crashes reported by CSGO.

Table 1: Previously Unknown Bugs Detected by CSGO.

Index	Source File	Bug Function	Bug Type
1	net/core/sock.c	sock_kmalloc()	null-ptr-deref
2	mm/backing-dev.c	max_bytes_store()	divide error
3	mm/backing-dev.c	min_bytes_store()	divide error
4	fs/hugetlbfs/inode.c	hugetlbfs_get_inode()	null-ptr-deref
5	fs/ext4/inode.c	ext4_writepages()	kernel panic
6	fs/ext4/inode.c	ext4_map_blocks()	deadlock
7	mm/page_alloc.c	current_gfp_context()	kernel panic
8	drivers/ptp/ptp_sysfs.c	max_vclocks_store()	kernel panic
10	kernel/locking/lockdep.c	down_write_nested()	deadlock
11	kernel/time/posix-timers.c	common_nsleep()	deadlock
12	mm/util.c	__kvmalloc_node_noprof()	kernel panic
13	kernel/rcu/rcutorture.c	rcu_sr_normal_complete()	kernel panic
14	kernel/workqueue.c	detach_worker()	use-after-free
15	mm/hugetlb.c	update_and_free_pages_bulk()	kernel panic
16	kernel/time/clockevents.c	clockevents_program_event()	deadlock
17	fs/ext4/extents.c	ext4_find_extent()	slab-out-of-bounds
18	drivers/acpi/acpica/utobject.c	acpi_ut_valid_internal_object()	slab-use-after-free
19	fs/ext4/inode.c	ext4_invalidate_folio()	kernel panic
20	fs/ext4/extents.c	ext4_ext_rm_leaf()	use-after-free
21	fs/ext4/extents.c	ext4_ext_binsearch()	use-after-free
22	net/ipv6/ip6mr.c	ip6mr_rules_exit()	kernel panic

Bug Statistics. As listed in Table 1, CSGO successfully identified vulnerabilities across multiple modules and subsystems within the Linux kernel. Among the identified bugs, 10 have been confirmed (bugs #1-4, 12-16, 22), with 8 already addressed and patched by maintainers (bugs #1, 4, 12-16, 22), and 2 are assigned CVE IDs #14 [18] and #22 [19]. It is worth noting that the file system module exhibited the highest concentration of vulnerabilities, totaling 7 cases (bugs #4-6, 17, 19-21), emphasizing its complexity and susceptibility to configuration-induced faults. Additionally, the memory management subsystem also displayed significant bug, contributing another 5 bugs (bugs #2, 3, 7, 12, 15). The detected vulnerabilities encompassed a wide spectrum of bug types, including kernel panics (bugs #5, 7, 8, 12, 13, 15, 19, 22), deadlocks (bugs #6, 10, 11, 16), and memory corruptions (bugs#1-4, 14, 17-18, 20-21), demonstrating the diversity and severity of bug uncovered.

Severity Analysis. Bugs in the Linux kernel can result in critical system failures. For instance, kernel panic bugs such as #7, 12, 15, and 19 typically stem from unsafe memory access or improper

resource handling, which can abruptly terminate the system and result in data corruption or loss. Use-after-free vulnerabilities like #14, 20, and 21 can lead to unpredictable behavior or privilege escalation, posing serious threats to the integrity of the system. Additionally, deadlock bugs, including #6, 10, and 16, can cause the kernel to hang indefinitely, rendering the system unresponsive and interrupting critical operations.

```
1  // configuration triggers the bug
2  static long set_virtual_ptp_ptp0_max_vclocks(long val) {
3      char command[256];
4      sprintf(command, "echo %ld >
           /sys/devices/virtual/ptp/ptp0/max_vclocks", val);
5      return system(command);
6  }
7  static ssize_t max_vclocks_store(struct device *dev, struct
        device_attribute *attr, const char *buf, size_t size) {
8      struct ptp_device *ptp = dev_get_drvdata(dev);
9      u32 max;
10     // This ensures 'max' is nonzero, but failed check upper
           limits.
11     if (kstrtou32(buf, 0, &max) || max == 0) {
12         return -EINVAL;
13     }
14     if (max == ptp->max_vclocks) {
15         return size;
16     }
17     ptp->max_vclocks = max;
18     ...
19     return size;
20 }
```

Figure 5: CSGO triggers a kernel panic in the driver module.

Bug Sample. We use bug #8 as an example to illustrate the effectiveness of the CSGO, as shown in Figure 5. In detail, this bug is triggered when user space writes an excessively large integer value to the `max_vclocks` sysfs entry in the Linux PTP subsystem, which originally lacked an upper bound check. Specifically, the function `max_vclocks_store()` receives the input string `buf`, which it then converts into an unsigned integer `max` using `kstrtou32()` (line 12). Although this conversion checks for errors or a zero value, there was no explicit check for an excessively large value, allowing an unrealistic configuration. As a consequence, the driver incorrectly accepts and stores this oversized value into the `ptp->max_vclocks` field (line 15). Following this step, the PTP driver tries to allocate memory and schedule operations based on the provided `max_vclocks` value, assuming it to be a legitimate user-defined number of virtual clocks. However, due to the absence of an upper limit validation, the kernel attempts to satisfy impractically large allocation requests. This scenario generates overwhelming memory pressure on the VM subsystem, causing allocation failures, unhandled exceptions, and ultimately leading to this bug.

CSGO can trigger this bug because first, it identifies the relationship between the configuration parameter `max_vclocks` and its associated PTP system call operations, and it generates test payloads that combine this correlated configuration modification with the relevant system calls. Specifically, by recognizing that modifying the `max_vclocks` setting (line 3) directly impacts the virtual clock handling logic (lines 11–13), CSGO can craft payloads to trigger this bug, by inducing the oversized memory allocation request and ultimately revealing the underlying kernel panic.

5.3 Overall Effectiveness Analysis

To answer **RQ2** and evaluate the effectiveness of CSGO, we compare the code coverage achieved and the number of detected unique bugs between CSGO and other existing tools. We implemented CSGO on top of Moonshine (denoted as CSGO-ms). By comparing CSGO and CSGO-ms with vanilla Syzkaller, Moonshine, and Actor, we demonstrate CSGO's fuzzing performance.

Table 2: Performance Comparison of CSGO, CSGO-ms, Moonshine, Syzkaller, and Actor.

Version	Moonshine	CSGO-ms	Syzkaller	CSGO	Actor
6.8	69108.2 (+28%)	88641.8	78085.6 (+23%)	96280.8	81822.8 (+18%)
6.10	67576.8 (+32%)	89476.4	84526.4 (+16%)	98320.0	93902.8 (+5%)
6.12	69135.2 (+33%)	92110.2	82625 (+21%)	100247.4	94721.2 (+6%)
6.13	68640.2 (+39%)	95202.2	82830.2 (+25%)	103778.0	94472.2 (+10%)
Average	68615.1 (+33%)	91357.7	82016.8 (+22%)	99656.6	91,229.8 (+9%)

Coverage Statistics. The detailed coverage statistics are shown in Table 2. As illustrated, CSGO achieves the highest code coverage across all evaluated kernel versions. Specifically, CSGO achieves 96280.8, 98320.0, 100247.4, and 103778.0 coverage on Linux kernels 6.8, 6.10, 6.12, and 6.13, respectively, with an average coverage of 99656.6. Compared with vanilla Syzkaller, CSGO demonstrates an improvement of approximately 22% on average. Additionally, CSGO-ms achieves 88641.8, 89476.4, 92110.2, and 95202.2 coverage across the corresponding kernel versions, averaging 91357.7, outperforming Moonshine by roughly 33%. Furthermore, CSGO outperforms Actor by an average of 9% across all kernel versions.

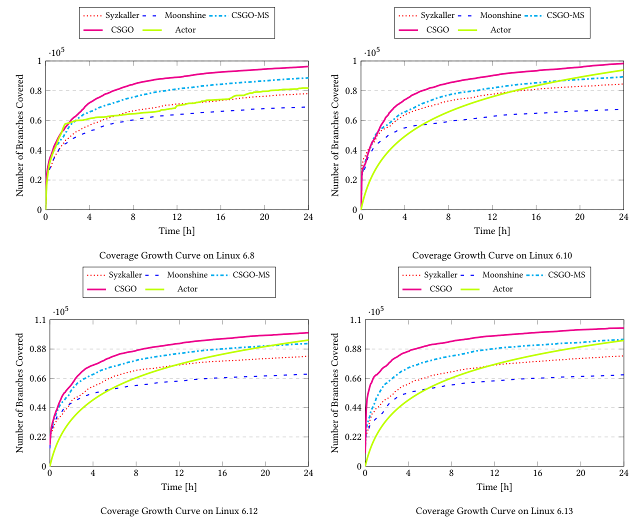

Figure 6: Coverage growth for CSGO, CSGO-MS, Syzkaller, Moonshine, and Actor for 24 hours.

Furthermore, we plot the coverage growth curves of CSGO in comparison to other fuzzing methods, depicted in Figure 6. From the graph, we observe an initial rapid increase in code coverage

for all evaluated fuzzing methods within the first 4 hours. After this period, the coverage growth rates begin to stabilize. However, throughout the fuzzing process, CSGO can grow faster and achieve higher coverage compared to other methods. This can be attributed to two main factors. First, CSGO can extract dynamic configuration information from the kernel and synthesize an enriched initial corpus consisting of configuration calls. This enriched corpus provides the fuzzing engine with high-quality seed inputs, allowing CSGO to efficiently explore kernel paths at an early stage, thus achieving higher coverage more rapidly. Second, during fuzzing, CSGO generates test payloads by combining system calls with configuration calls. By leveraging payloads that contain configuration-system call pairs, CSGO can trigger execution paths that are less frequently covered or unreachable by traditional fuzzing methods, ultimately achieving higher overall coverage.

Table 3: Unique Bug Number Detected by CSGO, CSGO-ms, Moonshine, Syzkaller, and Actor.

Version	Moonshine	CSGO-ms	Syzkaller	CSGO	Actor
6.8	2.6(+4.85×)	15.2	6.4 (+1.09×)	13.4	5.2 (+2.58×)
6.10	2.6 (+3.69×)	12.2	5.6 (+1.14×)	12.0	15.8(−0.82×)
6.12	2.4 (+4.83×)	14.0	6.6 (+1.09×)	13.8	12.6(+1.10×)
6.13	2.6 (+5.62×)	17.2	7.2(+1.38×)	17.2	17.4(−0.99×)
Average	2.6 (+4.75×)	14.7	6.5 (+1.19×)	14.1	12.7(+1.11×)

Bug Detection Statistics. We further compare the number of unique bug detected by CSGO, CSGO-ms, Moonshine, Syzkaller, and Actor, as shown in Table 3. As observed, CSGO and CSGO-ms detect an average of 14.1 and 14.7 bugs separately, outperforming Moonshine and Syzkaller by approximately 1.19 and 4.75 ×, respectively. Also, compared with Actor, CSGO and CSGO-ms achieve an average improvement of 1.11 × on average. Still, we find that Actor can detect more bugs than CSGO-ms on Linux kernel 6.10 and 6.13, this is due to the fact that Actor is designed to explore the kernel memory operation and therefore it is capable of detecting more memory-related bugs. These improvements demonstrate the efficacy of the configuration-sensitive fuzzing approach employed by CSGO, as combining configuration information with system calls can enable CSGO to trigger more diverse and complex kernel behaviors, leading to the discovery of a wider range of vulnerabilities.

5.4 Component-Wise Effectiveness Analysis

To answer **RQ3** and assess the individual contributions of each component to CSGO's ability to explore the kernel state space, we first compared the code coverage achieved and bug detected by CSGO, CSGO-NoMD, CSGO-ms, and CSGO-MsNoMD. By removing the mapping deduction component from both CSGO and CSGO-ms, we quantified the specific impact of this component on the overall fuzzing performance. Then, we sampled and analyzed the identified relations between configurations and system calls to evaluate the effectiveness of the mapping deduction component.

Coverage Statistics. The detailed coverage statistics in Table 4 provide a comparison of the code coverage achieved by CSGO and its variants. Compared to these variants, CSGO and CSGO-ms can achieve higher coverage, with an average improvement of 13% and

Table 4: Coverage Statistics of CSGO and Its Variants.

Version	CSGO-MsNoMD	CSGO-ms	CSGO-NoMD	CSGO
6.8	73454.6 (+6%)	88641.8 (+21%)	87212.4 (+12%)	96280.8 (+10%)
6.10	72564.0 (+7%)	89476.4 (+23%)	89011.6 (+5%)	98320.0 (+10%)
6.12	75056.8 (+8%)	92110.2 (+23%)	90742.2 (+10%)	100247.4 (+10%)
6.13	76476.4 (+10%)	95202.2 (+24%)	86140.4 (+4%)	103778.0 (+20%)
Average	74387.9 (+8%)	91357.7 (+23%)	88276.7 (+8%)	99656.6 (+13%)

23%, respectively. This improvement mainly comes from the mapping deduction component, which helps identify relevant combinations of configurations and system calls. By focusing on these relevant combinations, CSGO reduces the overall search space, avoiding unnecessary or redundant testing efforts. Consequently, it can generate more targeted and efficient test cases, improving fuzzing efficiency. This approach allows CSGO to quickly explore deeper or previously untouched kernel code paths, achieving higher overall code coverage in less time.

We further compare CSGO-NoMD and CSGO-MsNoMD with Syzkaller and Moonshine. Specifically, CSGO-NoMD shows an average coverage improvement of about 8% compared to Syzkaller and Moonshine. Because both CSGO-MsNoMD and CSGO-NoMD only differ from Syzkaller and Moonshine by having an enhanced initial configuration corpus, these results indicate the benefits provided by a better initial corpus. This improved corpus allows CSGO to explore kernel states more quickly and effectively from the beginning of the fuzzing process.

Bug Detection Statistics. We further compare the unique bug number and previously unknown bugs that were detected by CSGO, CSGO-MsNoMD, CSGO-ms, and CSGO-NoMD to quantify the impact of each component on the overall fuzzing performance.

Table 5: Unique Bug Number Detected by CSGO and CSGO-MsNoMD, CSGO-ms, and CSGO-NoMD.

Version	CSGO-MsNoMD	CSGO-ms	CSGO-NoMD	CSGO
6.8	13.4 (+4.15×)	15.2 (+0.13×)	11.8 (+0.84×)	13.4 (+0.13×)
6.10	11.4 (+3.48×)	12.2 (+0.07×)	11.8 (+1.11×)	12.0 (+0.02×)
6.12	12.8 (+4.33×)	14.0 (+0.09×)	13.2 (+1.00×)	13.8 (+0.05×)
6.13	16.2 (+5.23×)	17.2 (+0.06×)	14.0 (+0.94×)	17.2 (+0.23×)
Average	13.5 (+4.27×)	14.7 (+0.09×)	12.7 (+0.97×)	14.1 (+0.11×)

First, we evaluated how many previously unknown bugs listed in Table 1 could be identified by the variants CSGO-MsNoMD and CSGO-NoMD. Specifically, CSGO-MsNoMD was able to detect 13 of these bugs (bugs #1–9, 11, 15, 18, and 21). Meanwhile, CSGO-NoMD detected slightly more, finding a total of 16 bugs (bugs #1–9, 11, 15–18, 20, and 21). Next, we compared the number of distinct bug counts discovered by each variant (CSGO, CSGO-MsNoMD, CSGO-ms, and CSGO-NoMD). The detailed statistics are summarized in Table 5. From these results, CSGO-MsNoMD and CSGO-NoMD identified an average of 13.5 and 12.7 bugs, respectively. After incorporating the mapping deduction component, the variants CSGO-ms and CSGO detected slightly more bugs, improving by approximately 0.09 and 0.11 × on average. Moreover, compared to other fuzzing methods like Syzkaller and Moonshine, CSGO-NoMD achieved an average improvement of around 0.97 × in the

number of bugs detected. Similarly, CSGO-MsNoMD also showed an improvement of about 4.27 × on average over Syzkaller and Moonshine. These comparisons further highlight the benefits provided by the initial configuration corpus and mapping deduction in enhancing the fuzzing performance of CSGO.

Relevance Deduction Correctness. We further evaluate the accuracy of the configuration–system call relevance mappings generated by CSGO. In detail, we assess whether the relations learned by our weighted deduction are accurate across different kernel modules. In total, CSGO extracted 1,571 writable dynamic configuration entries from a Linux 6.13 kernel, while Syzkaller (`Commit 489e2d`) includes 4,499 system call descriptions, to minimize the search space, we sampled approximately 10% of configurations from each module, and we manually inspected the inferred relations and labeled any incorrectly associated system calls as false positives.

Table 6: Relevance Deduction Statistics for Kernel Modules.

Module	Total Configs	Sampled Configs	Learned Relations	Incorrect Relations	False Positive
Block Devices	59	6	242	11	4.55%
Drivers	298	30	4394	129	2.94%
File System	70	7	453	29	6.40%
Kernel	134	13	702	48	6.84%
Memory	124	12	132	7	5.30%
Network	886	89	7150	79	1.10%
Summary	1571	157	13073	303	2.32%

As shown in Table 6, among the 157 sampled configurations, CSGO inferred a total number of 13,073 configuration–system call relations, of which 303 were identified as incorrect, resulting in an overall false positive rate of 2.32%. Notably, the network and driver modules together account for the majority of learned relations, yet exhibit the lowest false positive rates, due to their well-defined interfaces and clearer access patterns. By contrast, modules such as file system, kernel, and memory show higher error rates, which attribute to complex internal dependencies, indirect access paths, and configuration semantics that vary by execution context. Results indicate that CSGO can capture relevant configuration–system call relations and maintain moderate accuracy.

6 Related Works

In this section, we review prior works related to kernel fuzzing and configuration-related testing techniques.

6.1 Kernel Fuzzing

Syzkaller [29], as the most well-known kernel fuzzer, has been integrated into the Linux kernel's CI/CD pipeline. By far, there are many works based on Syzkaller [12, 26, 27] that attempt to improve the quality of initial payloads. KSG [26] leverages runtime probes and path-sensitive static analysis to extract the kernel's submodule entries and corresponding arguments to generate test cases for certain kernel modules. SyzGen [4] automates system call specification inference for closed-source macOS drivers by refining system call knowledge and extrapolating dependencies from execution traces. SyzDescribe [12] extracts system call specifications by modeling invariant kernel driver contracts, capturing initialization procedures, and system call construction logic. These works primarily focus on analyzing the kernel's API interface and extracting system call information. In contrast, CSGO focuses on the dynamic configuration information and generates test cases that combine system calls with relevant configuration calls to explore more kernel state space.

There are works that target fuzzing for specific kernel subsystems. Hydra [6] provides modular primitives for file system fuzzing, enabling developers to build specialized checkers on top of a common framework. Saturn [34] focuses on the USB subsystem and introduces a host-gadget synergistic fuzzing approach to reveal bugs that emerge through cross-device interactions. KRACE [33] detects file systems's concurrency bugs by tracking thread information and modeling happens-before relationships across multithreaded system call sequences. StateFuzz [36] models kernel driver behavior as state machines and steers fuzzing based on detected state transitions, improving the discovery of semantic bugs in device drivers. BVF [28] generates eBPF programs that pass the verifier and uses runtime indicators to detect illegal behaviors, effectively turning verifier correctness checking into automatic discovery of logic bugs. Unlike these works that focus on specific subsystems or device interfaces, CSGO explores the kernel's dynamic configuration interface. Rather than targeting a fixed driver or protocol stack, CSGO discovers and test the kernel's dynamic configuration parameters exposed via virtual files, which influence control flow across a wide range of kernel components.

6.2 Configuration-Related Testing Techniques

Configuration is a critical factor in software systems, and its values can significantly impact the behavior and performance of these systems. Due to the complexity of the configuration space, many studies have aimed at improving the effectiveness of fuzzing by incorporating configuration information. ConfigFuzz [35] encodes program configuration options into fuzzable inputs, enabling simultaneous exploration of different configurations and input spaces during fuzzing. ECFuzz [17] employs multi-dimensional mutation strategies and unit-testing validation to explore complex configuration spaces in large-scale systems. ConFu [7] dynamically mutates runtime configurations at specific execution points to uncover vulnerabilities that manifest only under particular configuration conditions. CarpetFuzz [31] uses natural language processing techniques to extract and analyze relationships among configuration options from program documentation, enabling informed and constraint awared fuzzing. However, these works primarily focus on user-space applications and do not consider the unique challenges and complexities associated with kernel configurations.

For the Linux kernel, current works mostly leverage the kernel's static configurations to enhance fuzzing effectiveness or bug reproduction. For example, KernJC [22] uses patch-based version verification and graph-based configuration analysis to automate the construction of vulnerable environments necessary for kernel bug reproduction. Hasanov and Nagy [13] analyze and recommend kernel static configurations to maximize the inclusion of recent kernel patches, significantly improving the effectiveness of continuous fuzzing. However, static configuration mostly exposes compile-time or boot-time options that remain fixed for an entire boot; once the

kernel is up, those parameters never change, so each fuzzing instance can only exercise one point in a huge configuration lattice.

7 Discussion

Our current design of CSGO allows for some potential enhancements and improvements that can significantly increase its effectiveness further in finding deeply rooted bugs.

Leverage Static Configurations. Our current design primarily addresses dynamic configurations, those that can be modified at runtime and does not yet support static configurations set during compilation. Supporting static configurations introduces new challenges, such as analyzing kernel build-time options, resolving compile-time conflicts, and managing kernel image switching without interrupting fuzzing execution. These requirements pose both technical and logistical difficulties. Furthermore, many dynamic options are conditionally dependent on static ones. Therefore, future work can leverage both static and dynamic configurations to enhance the fuzzing efficiency. To improve completeness and configuration coverage, future work should aim to jointly analyze static and dynamic configurations.

Diversity of Configuration Values. While our current focus is on configurations with relatively simple value types, many kernel parameters are string-based and accept only a limited set of valid values. Unfortunately, these accepted values are often sparsely documented and must be inferred from kernel source code or scattered discussions, complicating automated extraction and validation. Given Linux's design philosophy, where source code often serves as the primary reference, this poses a significant barrier. In the future, we can address this limitation by incorporating hybrid fuzzing strategies, static analysis, and mining of runtime logs or documentation to identify valid string options, thereby enhancing both the depth and the breadth of configuration-sensitive fuzzing.

8 Conclusion

In this paper, we present CSGO, a kernel fuzzing framework that leverages dynamic configuration information to enhance the effectiveness of kernel fuzzing. CSGO identifies the kernel's dynamic configuration information and generates configuration calls as the initial corpus. During fuzzing, CSGO generates test payloads that combine both system calls and configuration calls. By leveraging the coverage as the indicator, CSGO can unveil the hidden relationship between extracted configurations calls and system calls, therefore generating test payloads focusing on configuration–system call combinations that are likely to trigger deeper or previously unexplored kernel paths. In our evaluation, CSGO identified 22 previously unknown bugs, with 10 confirmed, 8 fixed by the corresponding kernel maintainers, and 2 assigned CVE IDs. Moreover, CSGO can achieve a average of 21% higher code coverage compared to state-of-the-art kernel fuzzers.

9 Acknowledgments

We thank the shepherd and reviewers for their valuable comments. This research is partly sponsored by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62525207,62472448).

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, 421–432.

[2] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2023. No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions. doi:10.14722/ndss.2023.24688

[3] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *39th IEEE Symposium on Security and Privacy, SP 2018*. IEEE Computer Society, 711–725.

[4] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 749–763. doi:10.1145/3460120.3484564

[5] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *42th IEEE Symposium on Security and Privacy, SP 2021*. 677–693. doi:10.1109/SP40001.2021.00114

[6] Ellis Cohen and David Jefferson. 1975. Protection in the Hydra Operating System. *SIGOPS Oper. Syst. Rev.* 9, 5 (Nov. 1975), 141–160. doi:10.1145/1067629.806532

[7] Huning Dai, Christian Murphy, and Gail Kaiser. 2010. CONFU: Configuration Fuzzing Testing Framework for Software Vulnerability Detection. *International Journal of Secure Software Engineering* 1, 3 (2010), 41–55. doi:10.4018/jsse.2010070103 PubMed-not-MEDLINE.

[8] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. 2023. Fuzzing Embedded Systems using Debug Interfaces. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023 (ISSTA 2023)*. ACM, New York, NY, USA, 1031–1042. https://doi.org/10.1145/3597926.3598115

[9] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. 2023. ACTOR: Action-Guided Kernel Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5003–5020. https://www.usenix.org/conference/usenixsecurity23/presentation/fleischer

[10] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 91–100.

[11] Google. 2014. Kernel Address Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[12] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. 2023. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers . In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 3262–3278. doi:10.1109/SP46215.2023.10179298

[13] Sanan Hasanov, Stefan Nagy, and Paul Gazzillo. 2025. A Little Goes a Long Way: Tuning Configuration Selection for Continuous Kernel Fuzzing . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 521–533. doi:10.1109/ICSE55347.2025.00042

[14] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *NDSS*.

[15] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. doi:10.1145/3243734.3243804

[16] lcamtuf. 2013. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[17] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. 2024. ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Association for Computing Machinery.

[18] NVD. 2025. CVE-2025-21786. https://nvd.nist.gov/vuln/detail/CVE-2025-21786

[19] NVD. 2025. CVE-2025-38032. https://nvd.nist.gov/vuln/detail/CVE-2025-38032

[20] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[21] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1980–1997. doi:10.1109/TSE.2019.2941681

[22] Bonan Ruan, Jiahao Liu, Chuqi Zhang, and Zhenkai Liang. 2024. KernJC: Automated Vulnerable Environment Generation for Linux Kernel Vulnerabilities. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses* (Padua, Italy) *(RAID '24)*. Association for Computing Machinery,

New York, NY, USA, 384–402. doi:10.1145/3678890.3678891

[23] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[24] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. 2022. Tardis: Coverage-Guided Embedded Operating System Fuzzing. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 41, 11 (nov 2022), 4563–4574. doi:10.1109/TCAD.2022.3198910

[25] SimonKagstrom. 2015. KCOV. https://github.com/SimonKagstrom/kcov.

[26] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 351–366. https://www.usenix.org/conference/atc22/presentation/sun

[27] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. *HEALER: Relation Learning Guided Kernel Fuzzing*. ACM, New York, NY, USA, 344–358. https://doi.org/10.1145/3477132.3483547

[28] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. 2024. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program. In *EuroSys '24: Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys - Proceedings of the European Conference on Computer Systems)*. Association for Computing Machinery, United States, 689–703. doi:10.1145/3627703.3629562 Full text of this publication does not contain sufficient affiliation information. With consent from the author(s) concerned, the Research Unit(s) information for this record is based on the existing academic department affiliation of the author(s).; 19th European Conference on Computer Systems (EuroSys 2024) ; Conference date: 22-04-2024 Through 25-04-2024.

[29] Dmitry Vyukov and Andrey Konovalov. 2015. Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller

[30] Dmitry Vyukov and Andrey Konovalov. 2015. Syzlang: System Call Description Language. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md

[31] Dawei Wang, Ying Li, Zhiyu Zhang, and Kai Chen. [n. d.]. CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 1919–1936.

[32] Qinglong Wang, Runzhe Wang, Yuxi Hu, Xiaohai Shi, Zheng Liu, Tao Ma, Houbing Song, and Heyuan Shi. 2023. KeenTune: Automated Tuning Tool for Cloud Application Performance Testing and Optimization. Association for Computing Machinery, 1487–1490.

[33] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *41th IEEE Symposium on Security and Privacy, SP 2020*. 1643–1660. doi:10.1109/SP40000.2020.00078

[34] Yiru Xu, Hao Sun, Jianzhong Liu, Yuheng Shen, and Yu Jiang. 2024. Saturn: Host-Gadget Synergistic USB Driver Fuzzing . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 4646–4660. doi:10.1109/SP54263.2024.00051

[35] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing Configurations of Program Options. *ACM Trans. Softw. Eng. Methodol.* (2023).

[36] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3273–3289. https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-bodong

[37] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng