A Little Goes a Long Way: Tuning Configuration Selection for Continuous Kernel Fuzzing

Sanan Hasanov
University of Central Florida

Stefan Nagy
University of Utah

Paul Gazzillo
University of Central Florida

***Abstract*—The Linux kernel is actively-developed and widely-used. It supports billions of devices of all classes, from high-performance computing to the Internet-of-Things, in part because of its sophisticated configuration system, which automatically tailors the source code according to thousands of user-provided configuration options. Fuzzing has been highly successful at finding kernel bugs, being among the top bug reporters. Since the kernel receives 100s of patches per day, fuzzers run continuously, stopping regularly to rebuild the kernel with the latest changes before restarting fuzzing. But kernel fuzzers currently use predefined configuration settings that, as we show, exclude the majority of new patches from the kernel binary, nullifying the benefits of continuous fuzzing. Unfortunately, state-of-the-art configuration testing techniques are generally ill-suited to the needs of continuous fuzzing, excluding necessary options or requiring too many configuration files to be tractable. We distill down the needs of continuous testing into six properties with the most impact, systematically analyze the space of configuration selection strategies, and provide actionable recommendations. Through our analysis, we discover that continuous fuzzers can improve configuration variety without sacrificing performance. We empirically evaluate our discovery by modifying the configuration selection strategy for syzkaller, the most popular Linux kernel fuzzer, which subsequently found more than twice as many new bugs (35 vs. 13) than with the original configuration file and 12x more (24 vs. 2) when considering only unique bugs—with one security vulnerability being assigned a CVE.**

I. Introduction

Operating system kernels are among the most important layers of modern computing stacks, and the Linux kernel is one of the most actively-developed and widely-used kernels, used for web servers, Internet-of-Things devices, mobile phones, supercomputing servers, and more [1], [2], [3]. Linux supports so much variety because it is a highly-configurable software product line [4], [5], [6]. It has over 15,000 configuration options controlling architecture, memory management, scheduling, file systems, and much more. Users tailor the kernel to their needs by passing *configuration options* to its build system, automatically customizing the kernel [7], [8].

The Linux kernel's position in the global computing stack makes it a high-value target, with vulnerabilities routinely selling for millions of dollars [9]. Efforts to proactively find and fix kernel vulnerabilities have embraced *fuzzing*, an automated software testing technique that uncovers software bugs through the generation of massive amounts of random test cases. Kernel fuzzers feed random inputs to various kernel interfaces, such as system calls [10], [11] or device-specific I/O channels [12], [13]. Modern kernel fuzzers are fast [14], precise [15], [16], and support many interfaces [17], [18]. Google's syzkaller [10] has found nearly 4,000 vulnerabilities in the Linux kernel alone [19] and is consistently among the top reporters of kernel bugs [20], [21], [22].

The Linux kernel receives hundreds of changes *daily* in the form of code patches [23], [24]. To accommodate this rapid change, the syzbot test robot [25] runs syzkaller continuously during kernel development. While fuzzing success depends on running for as long as possible to maximize test coverage, syzbot must regularly stop fuzzing and update its kernel [26] to incorporate the latest changes, otherwise it risks missing new bugs introduced during development. To balance this trade-off, syzbot aims to run syzkaller for 12 hours before stopping to pull new changes and recompile the kernel [27]. Yet, the kernel is highly configurable, so whether code changes are *compiled in* depends entirely on whether the configuration file chosen by syzbot includes them.

But current configuration file selection strategies for kernel fuzzers *exclude* most code changes [25], [28]. For instance, syzkaller uses a small number of pre-selected configuration files [29], [30], builds each kernel variant, and fuzzes the variants separately. Fuzzing only a few configuration files is a reasonable strategy since continuous fuzz testers already have limited time to run before developers commit new changes. Moreover, fuzzers have specific requirements for configuration selection [31], [32]: they need to enable the kernel bug detectors, such as address sanitizers, and they must enable options required for booting kernels for them to fuzz. To set the thousands of other options, testers use Linux's default configuration file [29] to provide a minimal *bootable* kernel as a starting point [33].

Prior work, however, shows that Linux's default configuration file *excludes almost 80% of code changes from the compiled binary* [34], nullifying the effectiveness of continuous fuzzing. Unfortunately, state-of-the-art configuration generation techniques are generally ill-suited to the needs of continuous fuzzing. *Random configuration files* are rarely bootable (§ IV), leave up to 80% of kernel source code unreachable for fuzzers [35], and are unlikely to cover patches [34]. *t-wise sampling* [36], [37] and *combinatorial interaction testing* [38] ensure coverage of feature interactions but generate thousands of configuration files [37], which would require fuzzers to continuously build and test thousands of kernel binaries simultaneously instead of just a few, an unrealistic increase in resource requirements. *Maximal configuration* approaches attempt to build as many source lines as possible with a small number of configuration files [39], [40] but can be too

large to boot [41], [42], may fail to build [43], and require hours of build time [34], wasting limited time for fuzzing. *Configuration repair* modifies existing configuration files to ensure patch coverage [34], potentially helping continuous fuzzers avoid excluding recent changes. But being relatively new, configuration repair has only been applied to default configuration files for compile testing, not continuous fuzzing.

The key challenge is that configuration selection for continuous fuzzing has two conflicting goals, *fuzzer performance* and *configuration variety*. For performance, fuzzers want to use few configuration files, so the fuzzer can test for as long as possible on the same binary. But for variety, they want to use many configuration files, so that the fuzzer test can a larger variety of code in the kernel. Current kernel fuzzers emphasize fuzzer performance at the expense of configuration variety, which leads to missed opportunities to test most recent code changes and alternative variations of the kernel.

To help mitigate the challenge of continuously fuzzing highly-configurable software, we survey the space of configuration testing techniques and systematically analyze their impacts on continuous fuzzing. We identify six key properties of configuration testing strategies that impact a continuous fuzzer's performance, resource utilization, patch coverage, and ability to boot and run on the kernel. Based on our analysis, we create several recommendations for fuzz testers to select a configuration file based on their needs. Notably, we show how continuous fuzzers can improve bug finding capabilities without sacrificing performance or increasing resource utilization by using configuration repair to modify the configuration files used to build the kernel, all with no engineering changes to the fuzzer itself. To the best of our knowledge, no systematic analysis of configuration selection for continuous kernel fuzzers has been conducted previously and used to improve to their bug-finding capabilities.

We empirically evaluate our discovery by modifying syzbot's configuration selection strategy to include configuration repair. We emulate syzbot's test infrastructure and compare syzkaller's bug-finding capability and performance with and without configuration repair. Our experiments show a substantial impact on the number of previously-unreported bugs, with no significant impact on performance. With the modified configuration selection strategy, syzkaller found more than twice as many previously-unreported bugs (35 vs. 13) than with syzkaller's original configuration files, although 11 of these were found by both approaches. When considering only unique bugs, the new approach finds 12x more previously-unreported bugs (24 vs. 2). We reported all 35 new bugs found and are working with Linux developers to fix them. As of writing, 8 have been acknowledged, 4 have been patched. One bug exposed a security vulnerability that was assigned a CVE [44].

In this paper, we make the following contributions:

- We replicate prior patch coverage results on syzkaller's configuration selection tool, and show it fails to build more than half of a sample of recent patches (§ III).
- We study configuration selection strategies, identify the key properties that impact continuous fuzzing and provide recommendations for testers (§ IV).
- We modify syzkaller's configuration generation to include configuration repair and compare its new bug finding and performance results with and without repair (§ V).
- We reported 35 previously-unreported kernel bugs to Linux developers and one security vulnerability (§ VI).

II. Background

This section provides an overview of kernel fuzzing and kernel configurability.

A. Coverage-Guided Kernel Fuzzing

Fuzzing is one of today's most successful software bug discovery techniques, having found thousands of bugs and vulnerabilities in numerous applications, kernels, and other computing systems. Given a software target, fuzzers operate by generating massive amounts of random test cases and executing each on the target, leveraging lightweight program introspection to detect interesting *test-case-induced* program states (e.g., crashes [45], time-outs [46], or invariant violations [47]). Most fuzzers today are *coverage guided*: instrumenting the target to report the code coverage of all test cases, but saving—and subsequently mutating—only the minority that reveal *new* code coverage. Popular coverage-guided application fuzzers include AFL++ [45] and libFuzzer [48].

With the success of application fuzzing, many efforts are extending coverage-guided fuzzing to OS kernels. Like application fuzzers, kernel fuzzers operate by feeding test cases to the kernel's various input vectors: general-purpose system calls [10], [11], driver-specific I/O control handlers [15], [49], or the hardware–kernel memory boundary (e.g., memory-mapped I/O, port I/O, direct memory access) [12], [13], [18], [17]. By far today's most popular kernel fuzzer is Google's syzkaller [10], which has found over 10,000 bugs in numerous kernels such as Linux, FreeBSD, OpenBSD, and Android. Many academic and industrial efforts are continuing to improve syzkaller to attain higher speed [14], generate more precise inputs [16], and find non-trivial classes of bugs [50].

B. Highly-Configurable OS Kernels

TABLE I: Recent operating system kernels and their configuration specification languages alongside their estimated total configurable features. We compute each by searching for configuration option names in their respective configuration specification languages.

Kernel	Config. Language	Approx. Configs
Android	Kconfig [51]	19,392
FreeBSD	config [52]	1,440
Linux	Kconfig [53]	19,397
NetBSD	config [54]	3,141
OpenBSD	config [55]	1,252
XNU	config [56]	1,004

Modern OS kernels are veritable software *product lines*: codebases intentionally developed to be compilable as a multitude of unique *variants*. For everyday kernels, variants often take on the form of OS-specific distributions (e.g.,

Ubuntu and Debian, which are derived from the Linux kernel [53]). However, the highly-configurable nature of today's kernels enables fine-grained control over virtually all build characteristics—from toggling-on specific features (e.g., TLS, IPV6), the desired target architecture (e.g., ARM, X86-64), security mechanisms (e.g., KASLR, KASAN), and much more.

```
1 CONFIG_WERROR=y   # Always installed.
2 CONFIG_SYSVIPC=n # Never installed.
3 CONFIG_AUDIT=m    # Optional module.
```

Listing 1: An example synthetic KCONFIG feature configuration.

To enable modular configuration, today's kernels are often equipped with *feature-modeling systems*. Table I shows the feature-modeling systems and approximate feature counts for various commodity kernels, with the most popular being KCONFIG [53]. At a high level, a KCONFIG configuration governs any number of kernel features, with each represented as "`CONFIG_FEATURE = y` (always installed) `||` `n` (never installed) `||` `m` (an optional loadable module)". Listing 1 displays a synthetic example of a KCONFIG kernel configuration.

While facilitating a seemingly-endless number of configurations is important for maintaining broad deployment flexibility, subtle interactions between kernel features often bear unforeseen consequences. In many cases, specific feature combinations create unbootable or otherwise unusable builds; while others introduce obvious security flaws (known as "misconfiguration" bugs [57]). The configurability of kernels complicates fuzzing because testers must first choose a configuration file to compile the kernel binary. But the configuration file determines what source code is included in the binary. With so many possible options, ensuring the binary even has the desired code to test is non-trivial, much less ensuring the code is tested.

III. PATCH COVERAGE OF FUZZER CONFIGURATION FILES

A study of *configuration repair* for patch coverage found that commonly-used configuration files for testing exclude most patches to the kernel codebase [34]. Configuration repair in this context refers to a technique that automatically modifies a configuration file to change what code the configuration file covers. For instance, krepair [34] takes a Linux kernel patchfile (or any set of files and lines) and a configuration file and automatically discovers and applies changes to the configuration file so that the code in the patchfile is covered when building the repaired configuration file. Since kernel configuration files are applied at build time, any code not covered is excluded from the kernel binary. For continuous fuzzers, when a configuration file it uses to build a kernel excludes patched code, no matter how much run-time coverage the fuzzer is able to achieve, it will never be able to cover the patches excluded during the build. Yet including large amounts of code results in unbootable or slow kernels [34].

Prior work evaluating coverage limitations of configuration files used in testing only evaluated Linux's default configuration `defconfig`. But syzkaller and other fuzzers add additional configuration options to `defconfig`, potentially leading to more code included in the kernel binary than with `defconfig`. To evaluate how much code kernel fuzzers include, we replicate the prior configuration file patch coverage experiment by modifying its artifact [58]. Instead of `defconfig`, we evaluate the configuration files from two popular kernel fuzzers, syzkaller and kAFL, to see how much patched code they omit. Our modified scripts are available in our publicly-available artifact [59].

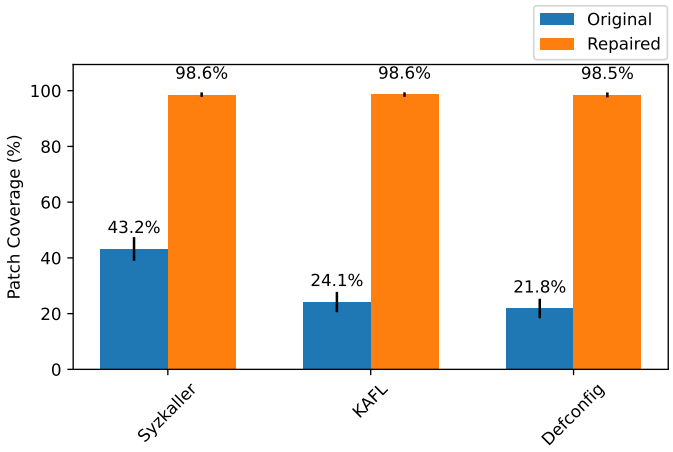


Fig. 1: Average patch coverage of syzkaller, kAFL, and defconfig.

.

To compare to existing work, we use an identical experimental setup. As previously reported, there are 507 randomly-selected patches from a "whole year (2021/09/19–2022/09/18) of Linux kernel development, which provides a 5% margin of error with a 98% confidence level" [34]. We use syzkaller's own configuration generation tool, `syz-kconf` [29] on the committed version of the kernel for each patch. kAFL, in contrast, provides a fixed configuration file [32], [28], which we use for each committed version of the kernel.

Figure 1 shows the results of replicating the patch coverage experiments on syzkaller and kAFL's configuration files in the first two bar groups. The last bar group is patch coverage results for `defconfig` as taken from the original study. The y-axis is the average patch coverage across all patches. For each bar group, the first bar is the original configuration file's average patch coverage. The second is the patch coverage after applying configuration repair with the `krepair` tool [34]. The error bars represent the 95% confidence interval of the average.

Prior work showed `defconfig` only covers 21.8% of patched code on average, while after applying `krepair`, the resulting configuration file covered 98.5%. Comparatively, syzkaller's `syz-kconf` covers more, 43.2%, although it is still less than half of patched code. kAFL's configuration file, however, is similar to `defconfig`, with 24.1% patch coverage. `krepair` increases patch coverage to 98.6% for both syzkaller and kAFL. The error bars show the patch coverage averages of the sample demonstrate a statistically significant improvement to patch coverage; the repaired configurations are over 98% in

all three cases with a very small margin of error. Although we evaluate syzkaller only in this paper, given the improvements of patch coverage for both syzkaller and kAFL, we expect the results to apply to kAFL and other kernel fuzzers.

Summary: Popular kernel fuzzers select configurations that exclude the majority of patched code—counteracting the benefits of continuously fuzzing updates to the kernel codebase.

IV. Configuration Selection Strategies for Continuous Kernel Fuzzing

We first discuss existing configuration selection techniques and weigh their trade-offs with respect to continuous kernel fuzzing. Next, we distill the needs of configuration selection for continuous fuzzing into six properties and analyze each technique across these categories. Finally, we provide recommendations for configuration selection strategies to guide designers of continuous fuzzing platforms for picking the best strategy or combinations of strategies for their goals.

A. Configuration Testing Techniques

There are many techniques for generating configuration files for testing, each of which achieves specific testing goals. We categorize them into seven groups of techniques.

Hand-selected configurations aim to customize kernels for unique deployments. For instance, syzkaller and kAFL define hand-selected configurations enabling options required for booting their respective test infrastructure [29], [28]. As shown in § III, these configurations exclude most patches, making them insufficient for continuous testing of kernel changes.

Default configurations are configurations shipped with the Linux kernel [40] and serve as the basis for building customized, hand-selected configuration files [33]. They are small, enabling relatively few features. *Minimal configurations* disable as many features as possible. Minimal configurations can be used as a fast-building sanity check recommended, for instance, in the Linux patch submission guidelines [60]. *Maximal configurations* enable as many features as possible to build as much code as possible, though not necessarily all; e.g., Linux's `allyesconfig` enables about 80% of the codebase [39]. Commonly-used for build testing, maximal configurations have also been used for static analysis [39] and testing [37]. Maximal configuration have little variety, since the goal is maximizing coverage, not feature interactions.

Random configurations, in contrast, are created by randomly assigning options to be either enabled or disabled, such as with the Linux kernel's `randconfig` tool [40]. Randomly generated configurations, we find, are not usually bootable, but they are used for build testing and static analysis [34]. *Interaction testing* approaches improve on random testing by maximizing coverage of *feature interactions*, i.e., combinations of configuration options. Many algorithms have been developed for this purpose [37], including t-wise [36], [37] sampling and combinatorial interaction testing [38]. These algorithms are effective at testing a wide variety of configurations, but they also require testing a very large number of configuration files, making them less suited to continuous fuzzing.

Configuration repair is not a configuration generation technique per se, but a technique to automatically modify existing configuration files. Configuration repair has only been applied to default configuration files to ensure patch coverage [34], but the approach theoretically works to modify any configuration file. Repair can preserve most options from the original file, giving it the potential for preserving a continuous fuzzer's required options for booting and testing a kernel.

B. Considerations for Continuous Fuzzing

Prior configuration testing techniques have been applied to a range of software analyses, including running tests, build testing, and static analysis. To our knowledge, prior work has not explicitly addressed the needs of continuous fuzzing, which has several unique properties distinguishing it from other testing techniques.

Bootable: Fuzz testers that rely on executing tests need the kernel to be bootable on the test infrastructure. Otherwise, the kernel cannot be tested. *Required options:* Kernel fuzz testers rely on the kernel's own bug detectors to report certain warnings, such as address sanitizers. These bug detectors are enabled via configuration options [31]. If not present, the fuzzer cannot identify bugs. *Patch Coverage*: As prior work [34] and our replication study shows, current continuous fuzzing approaches, which use hand-selected configuration files, miss the majority of patched code, contravening the core goal of testing continuously on new code changes. Fuzzers require at least configuration settings that include new code changes into the kernel binary for continuous testing.

Few Files: Fuzzing depends on high test execution throughput to cover as much of a codebase as possible, so fuzzers are typically left to run for as long as feasible [27], [61], [62], [19], [63], hours, days, or even weeks at a time. This need complicates configuration testing, because many prior techniques, such as feature interaction testing, depend on generating and testing thousands of configuration files for highly-configurable software like Linux [37], each of which needs to be fuzzed individually for as long as possible. Therefore, configuration testing strategies for continuous fuzzers are restricted to producing few configuration files in to preserve the resource utilization of the current practice of using a single hand-selected configuration file.

Variety: While fuzzers may not be able to add much configuration variety to testing each new version of the kernel, they can still add variety by altering the configuration file between runs on new versions. As Abal et al. [65] reveal, modern-day OS kernels such as Linux contain a significant number of security vulnerabilities dependent on multi-feature interactions. For example, Table II showcases 10 recent configuration-dependent vulnerabilities in the Linux kernel found in the CVE database [64], their assessed severity, and their respective configuration features. For example, reaching vulnerabilities CVE-2021-28039 and CVE-2021-20194 require the exclusion of kernel features `XEN_BALLOON_MEMORY_HOTPLUG` and

TABLE II: Recent configuration-dependent vulnerabilities and their relevant configuration features. Features **in red** were *not* present in any kernel configurations fuzzed by SyzKaller at the time the vulnerability was reported.

CVE [64]	Severity	Kernel Configuration Features
CVE-2023-4155	5.8 (Med)	VMAP_STACK
CVE-2023-3090	7.8 (High)	IPVLAN
CVE-2023-0461	7.8 (High)	TLS ‖ XFRM_ESPINTCP
CVE-2021-35039	7.8 (High)	!MODULE_SIG
CVE-2021-28039	6.5 (Med)	!XEN_BALLOON_MEMORY_HOTPLUG && XEN_UNPOPULATED_ALLOC
CVE-2021-20194	7.8 (High)	BPF && BPF_SYSCALL && CGROUPS && CGROUP_BPF && !HARDENED_USERCOPY
CVE-2018-19854	4.7 (Med)	CRYPTO_USER
CVE-2018-17182	7.8 (High)	DEBUG_VM_VMACACHE
CVE-2017-8070	7.8 (High)	VMAP_STACK
CVE-2017-7889	7.8 (High)	STRICT_DEVMEM

HARDENED_USERCOPY, respectively. Therefore kernel fuzzing, if it hopes to uncover such configuration-dependent bugs, needs at least some variety in its configuration file selection.

Fast Build: Critical to a fuzzer's bug-finding effectiveness is its ability to maintain a high test case throughput, requiring each component of the fuzzing loop to be as optimized as possible. Application-level fuzzers like AFL [45] and libFuzzer [48] have long benefited from performance enhancements in their code coverage tracing [66], [67] and process execution [68], [69] steps, which make up the two most time-consuming components of application fuzzer designs [70]. Unfortunately, current kernel fuzzers have significantly more complex components—relying on virtualized and/or emulated environments to *run* the kernel—that cannot easily be decomposed and optimized for higher fuzzing speeds. Yet, compared to application fuzzers, kernel fuzzers aim to fuzz software that is substantially larger in size; for example, the Linux v6.6 kernel is upwards of over half of a gigabyte large. Larger kernel configurations create larger kernel builds, placing higher resource strain on a fuzzer's virtualization and emulation infrastructure. Therefore, kernels using smaller kernel binary have the potential for better performance and also require less time to build, saving more time for fuzzing.

C. Recommendations for Configuration Selection

Table III summarizes the impact of configuration testing techniques on continuous fuzzing across the six properties identified above, one per column. Each row marks which properties each technique has.

Considering bootability, in our tests, `allyesconfig` fails to boot in syzkaller's QEMU-based virtualization setup [31], which is not surprising given prior works' conclusions that it can be too large to boot [41], [42] or may even fail to build [43]. `allnoconfig` also failed to boot in our tests. Additionally, generating 100 `randconfigs` resulted in no bootable kernels. `defconfig` is bootable, but lacks the options required by the fuzzer, which is why syzkaller's syz-kconf tool adds additional options to `defconfig`.

Three techniques remain: hand-selection, interaction testing, and repair. Hand-selection fails to include changes most of the time (§ III)—nullifying the benefits of fuzzing continuously and also adding no configuration variety. Interaction testing requires generating so many configuration files that only increasing the compute resources considerably would make it feasible. Moreover, not all generated configuration files will be bootable or cover patches, requiring manual effort to filter out unusable configurations. Computational resource requirements can be computed by by multiplying the number of configurations used by the resources needed by one fuzzing run. Resources needed for interaction testing are proportional to the number of unique configuration files it generates, as each one requires building and fuzzing a separate kernel binary.

In contrast, configuration repair does achieve patch coverage, even with only a single repaired configuration file. This avoids the computational resource burden of interaction testing and, as our evaluation (§ V) shows, generally preserves build time and the required options needed for fuzzing. As a trade-off to preserving configuration settings, however, it only adds relatively small amounts of variety, though our evaluation shows a substantial impact.

We make the following recommendations for adding configuration testing techniques to continuous fuzzing:

Repair hand-selected configuration files for continuous fuzzing. Continuous fuzzers should use hand-selected configuration files that have been repaired, since this will use the same compute resources as hand-selected, but add the benefits of patch coverage and configuration variety.

Use interaction testing if adding compute resources is viable. For platforms willing to add considerable compute resources to support fuzzing many configuration files in parallel, interaction testing can provide more variety than repairing hand-selected configuration files.

Use hand-selected configurations when only concerned with a specific device. For fuzz testers who only need to test a specific configuration of the kernel, e.g., device manufacturers who are only concerned with specific kernel features, variety is unnecessary, and a hand-selected configuration file suffices.

Use interaction testing if only concerned with a specific kernel version. For fuzz testers only concerned with a specific kernel version, i.e., a long-term release version [71], continuous testing is not necessary. Instead, the tester can add variety with interaction testing or random testing.

V. Evaluation

We evaluate the effects of applying our recommendation to use configuration repair for syzkaller's current hand-selected configuration file approach. We emulate syzbot's test infrastructure that runs syzkaller, but modify syzbot's configuration generation process to include additional variety using the krepair configuration repair tool [72].

Besides configuration selection, all other design decisions are left in place. We compare syzkaller's performance and bug-finding capabilities with and without adding configuration repair.

A. Experimental Setup

We first selected a set of previous syzkaller runs, since the configuration files used are recorded in bug reports on

TABLE III: A comparison of configuration selection strategies for the needs of continuous fuzzing

Selection Strategy	Techniques	Bootable	Required Opts	Patch Coverage	Few Files	Variety	Fast Build
Repair	krepair [34]	✔	✔	✔	✔	✔	✔
Hand-Selected	syzkaller [29], kAFL [28]	✔	✔	✗	✔	✗	✔
Interaction Testing	t-wise [36], [37], combinatorial [38]	✔	✔	✔	✗	✔	✗
Default	`defconfig` [40]	✔	✗	✗	✔	✗	✔
Maximal	`allyesconfig` [40], vampyr [39]	✗	✔	✔	✔	✗	✗
Minimal	`allnoconfig` [40]	✗	✗	✗	✔	✗	✔
Random	`randconfig` [40]	✗	✗	✔	✗	✔	✗

syzkaller's reporting dashboard [25]. We took a sample of 30 such configuration files from these previous bug reports. Then, to get Linux versions close to those that syzkaller was testing at the time of the report, we check out one or more commits from the Linux development tree [23] from the time period given in the bug report that provided the configuration file, for a total of 46 unique commits. The set of Linux commit IDs[1], configuration files[2], and URLs of the bug reports from the syzbot dashboard[3] can be found in the publicly-available artifact [59].

For each combination of configuration file and Linux commit, we ran syzkaller with and without the krepair-modified configuration file. To repair the configuration files, we checked out each Linux commit and used krepair to modify the original configuration file, feeding the patch file that updated the version to the commit ID, i.e., with `git show`, as input to krepair. We configured, built, and ran syzkaller twice for each commit/configuration combination on the same commit IDs, once with the original configuration and once with the modified configuration files. For both sets of configuration files, we run syzkaller using the same virtualization settings as described in its documentation [31], QEMU with 4GB RAM and 8 virtual CPUs per run, and for the same amount of fuzzing time, 12 hours as described by the syzkaller developers on their mailing list [27].

All experiments were run on a server with a 2.25GHz AMD EPYC 7742 and 512GB of RAM, running Ubuntu 22.04.2 LTS. Our publicly-available artifact contains the experiment scripts used to run the experiments [59].

B. Research Questions

To evaluate the impact of modifying syzbot's configuration selection, we ask the following research questions.

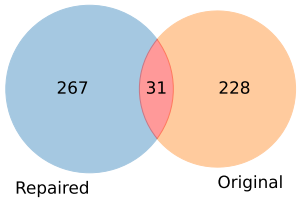
RQ1 (Bug discovery) How does configuration repair affect bug discovery?

RQ2 (Performance impacts) How does configuration repair affect performance?

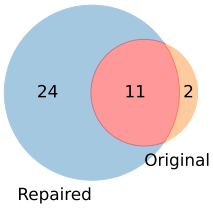
RQ3 (Configuration variety) How much configuration variety is introduced by configuration repair?

C. RQ1: Bug Discovery

To evaluate bug-finding capacity, we measure the number of alarms found by syzkaller in all runs, both with and without the krepair-modified configuration file. Since the same bugs are reported multiple times in the same run and across runs, we deduplicate and aggregate the bugs found. After deduplication, syzkaller found 269 bugs with the original configuration files and 298 with the krepaired configuration files.



(a) All bugs found.



(b) Previously-unreported bugs found.

Fig. 2: Bugs found by syzkaller using krepaired configuration files compared to the original configuration files.

Even though the same set of kernel versions and the same amount of total fuzzing time was used, there was surprisingly little overlap in the bugs found when comparing the two configuration file selection approaches, as shown by the Venn diagram in Figure 2a. There were only 31 bugs that were found by both approaches. This suggests that additional configuration variety affected which bugs the fuzzer was able to find in the same amount of time.

To find previously-unreported bugs, we search by hand syzbot's bug reporting history [25] for the same bug. While both approaches found a similar number of bugs, most of the bugs had been previously reported. When considering only new, previously-unreported bugs, syzkaller with configuration repair found considerably more, 35 new bugs compared to only 13 with the original configuration files as shown in Figure 2b. Moreover, since 11 new bugs were found by both, syzkaller with configuration repair found 24 unique *new* bugs compared to only 2 with the original configuration files—a 12x increase. § VI lists all new bugs found.

We analyzed the types of all bugs found by syzkaller with

[1] icse25-master/data_tables/Table_of_all_crashes.xlsx
[2] icse25-master/camera_ready/configuration_files/
[3] icse25-master/links_to_syzbot_bug_reports.txt

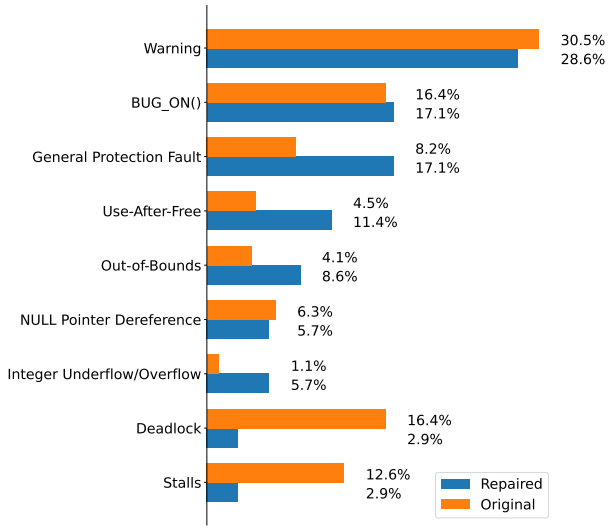


Fig. 3: Comparison of the distribution of bugs found repaired and original configuration files.

both the krepaired and original configuration files. Figure 3 shows the percentage of bugs per category for the krepaired configuration files, which yielded 298 bugs, and the original, which yielded 259. Warnings are bugs caused by violations of assertions written into code by developers. Such bugs are reported as warnings because developers prefer to trigger a warning and continue running rather than a kernel panic to, e.g., ease debugging and recovery of the kernel [73], although users may opt to panic on assertion violations [74]. `BUG_ON` bugs are assertion violations that developers decide should panic the kernel [75]. General protection faults, uses after free, out-of-bounds accesses and NULL pointer dereferences, integer overflow/underflow, stalls, and deadlock are from various kernel sanitizer and bug-reporting tools [76] whose output is recognized by syzkaller.

The types of bugs found both with and without the krepaired configuration file are comparable, as shown. Overall, we observe that the majority (54%) of bugs found using the krepaired configuration with syzkaller are memory safety bugs, which are highly relevant to kernel security. A total of 17.1% are classified as temporal memory use-after-frees (11.4%) and null-pointer dereferences (5.7%). Other memory safety errors include out-of-bounds accesses (8.6%), generic protection faults (17.1%), and integer-related errors (5.7%). We also see that 5.8% of found bugs fall under stalls or deadlocks, which are also commonly-exploited bug classes [77], [78].

RQ1: Modifying the configuration selection strategy helped the fuzzer identify different bugs and more previously-unreported bugs compared to the original configuration files.

D. RQ2: Performance Impacts

We evaluate the performance impacts of repairing configuration files before fuzzing by rerunning the steps of the bug-finding experiments, measuring (1) configuration generation time, (2) kernel build time, and (3) fuzzer throughput and coverage. We randomly-selected 10 Linux commits, which can be found in the artifact [59], to perform the evaluation and used the same original and krepaired configuration files used in the bug-finding evaluation.

TABLE IV: Five-point summaries of kernel configuration time in seconds.

	Original Configurations	Repaired Configurations
Min	2.23	39.48
25th	2.38	43.53
Median	2.39	44.27
75th	2.43	170.11
Max	2.57	172.99

TABLE V: Five-point summaries of kernel build times in seconds.

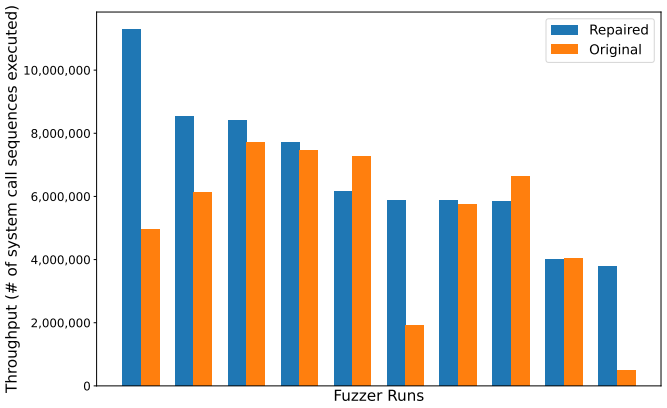
	Original Configurations	Repaired Configurations
Min	120.47	105.82
25th	154.64	193.17
Median	180.29	261.65
75th	240.25	270.96
Max	251.42	412.82

Configuration times: Table IV summarizes the distributions of configuration times for both the original and krepaired configuration files. To configure the kernel with an existing configuration file, the user runs `make olddefconfig`, which imports an existing configuration file, checking it for consistency. This process typically takes only seconds, as shown in the five-point summary for the original configuration. For the krepaired configuration files, we first run krepair on the configuration file before importing it, which can take several minutes. Although the krepair time was not included in the 12-hour fuzzing time in the bug-finding experiments, krepair time is very small compared to the typical fuzzing times.

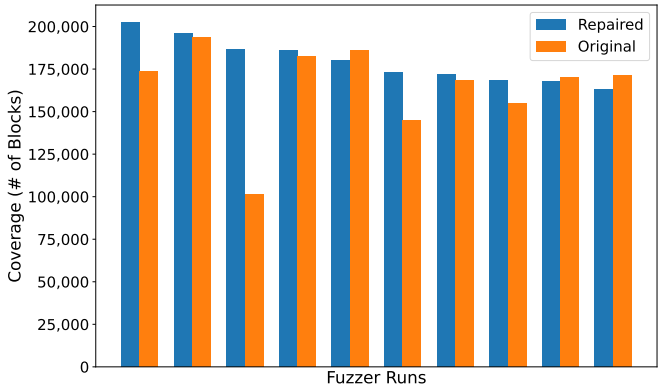
Build times: Table V summarizes the distributions of build times for the original and krepaired configuration files. Once configured, we build each kernel, parallelized with `make -j`, measuring its time. Given the small configuration files and high core-count of the experiment machine, build times were less than 10 minutes in all cases.

Fuzzer performance: To evaluate fuzzer performance, we measure test case test case throughput and total code coverage, which are recorded by syzkaller. Test case throughput is the number of test cases generated and executed over the 12 hours. High throughput is desirable for fuzzing, because it can increase the opportunities to find bugs. Total code coverage is the total number of basic blocks reached by any test case. Code coverage is critical to fuzzers' aim of testing as much of the codebase as possible. For comparing throughput, we follow the procedure of prior fuzzing literature [79] and report the total test cases (i.e., system call sequences) executed per trial; and for code coverage, we report the fuzzer-logged counts of kernel basic blocks reached. We discuss our results below.

Test Case Throughput: Figure 4a compares test case throughput for syzkaller with and without configuration repair for each fuzzing trial. We posit that including more kernel code likely causes fuzzing to execute more kernel code at the cost of some throughput. On average, we observe that fuzzing on



(a) syzkaller throughput with the original and krepaired configuration.



(b) syzkaller coverage with both the original and krepaired configuration.

Fig. 4: Comparing syzkaller performance before and after using krepair.

krepaired configuration files results in average throughput that is 28.75% higher. In 3 of 10 trials, however, syzkaller achieves a higher throughput with its original configuration file. The krepaired configuration files only have an average of 76 kernel more features than the original. Given the small amount of change made to the configuration files, the throughput results do not reveal a marked difference in throughput between the original and krepaired configuration file.

Code Coverage: High fuzzing throughput alone is not necessarily better for fuzzing if it is caused by repeatedly covering the same fast-executing code paths. Coverage is just as important for reaching bugs. We therefore also compare the code coverage of syzkaller with and without repaired configuration files. Following standard practice in fuzzing [50], we measure code coverage in basic blocks. Figure 4b shows the per-trial block coverage of syzkaller before and after using krepair.

In 3 of 10 trials, syzkaller achieves slightly higher code coverage with its original configuration file. Yet for the remaining 7 of 10 trials syzkaller coverage is higher with the repaired configuration files and by larger margins. Overall, coverage averages 9.04% more basic blocks than with the original configuration file. With a higher average code coverage and throughput, we can conclude that using krepair to modifying the configuration selection strategy does not harm the performance of fuzzing. Indeed, the increase configuration variety and build coverage of code changes is likely the source of improved bug finding, since it opens new code paths to exploration and provides evidence for why there was discovery of previously-unreported kernel bugs in § V-C.

TABLE VI: Number of configurations options changed by krepair.

Min	25th	Median	75th	90th	Max
57	70	76	90	115	1264

RQ2: Fuzzing with repaired kernel configurations upholds similar fuzzing performance to the original unmodified configurations.

E. RQ3: Configuration Variety

We measure how much configuration variety using krepair introduces when ensuring patch coverage by counting the number of options in the configuration file before and after repair. Table VI shows the distribution of the number of options changed as a five-point summary, i.e., quartiles, the minimum, and the maximum, plus the 90th percentile to show how the vast majority of cases behave. In most cases (90th percentile), there were 115 or fewer changed options, representing a very small change in the configuration file. While the maximum was 1,264 options, the kernel has over 19,000 options available in the Linux kernel, representing less than a 7% change in the file.

While even this modest amount of configuration variety had a surprisingly large increase bug-finding capabilities, it also helps explain why fuzzing performance had little impact. The kernel binaries produced were very similar to the original configuration file. This similarity also ensured that the kernel was still bootable in syzbot's testing infrastructure.

RQ3: Variety in configurations improves bug-finding capabilities of fuzzing while preserving the performance, bootability, and other configuration requirements of the fuzzer.

VI. Previously-Unreported Bugs

Table VII lists all 35 previously-undiscovered bugs identified by syzkaller when using repaired configuration files. The table shows the type of bug (separated by dashed lines), the function it was found in, the commit ID of the kernel, whether we could reproduce the bug, whether developers confirmed it, and whether it has been patched. All 35 bugs have been reported to Linux developers. As of the time of writing 8 have so far been confirmed and 4 have been patched. Moreover, we discovered an out-of-bounds access in `fbcon_set_font` that causes a local denial-of-service vulnerability that was assigned a CVE [44] with a medium severity score of 5.5.

TABLE VII: All previously-unreported bugs found by syzkaller when using the modified configuration selection strategy.

ID	Type	Location	Kernel Version	Reproducible	Confirmed	Patched
1	Use-After-Free	hci_conn_hash_flush	b7455b10da762f2d		✔	
2	Use-After-Free	sco_sock_timeout	b7455b10da762f2d			
3	Use-After-Free	f2fs_iget	09e41676e35ab06e	✔		
4	Use-After-Free	diFree	09e41676e35ab06e			
5	Null Ptr Deref	filemap_fault	b7455b10da762f2d			
6	Null Ptr Deref	ext4_update_overhead	b7455b10da762f2d			
7	Out-of-Bounds Access	extAlloc	b7455b10da762f2d	✔		
8	Out-of-Bounds Access	fbcon_set_font	a54df7622717a40d	✔	✔	✔
9	Out-of-Bounds Access	f2fs_iget	509583475828c4fd	✔		
10	Out-of-Bounds Access	f2fs_iget	66eee64b235411d5	✔		
11	Out-of-Bounds Access	ntfs_test_inode	a54df7622717a40d			
12	Protection Fault	nl802154_trigger_scan	4d6d7ce9baaf9e67	✔	✔	✔
13	Protection Fault	efivar_lock	9fbee811e479aca2	✔	✔	
14	Protection Fault	floppy_ready	9ce08dd7ea24253a	✔		
15	Protection Fault	blkg_destroy_all	09e41676e35ab06e			
16	Protection Fault	reset_interrupt	b7455b10da762f2d	✔		
17	Unspecified Bug	page_add_anon_rmap	a54df7622717a40d	✔	✔	✔
18	Unspecified Bug	rcu_core	465461cf48465b8a		✔	
19	Unspecified Bug	smp_call_function	b7455b10da762f2d	✔	✔	
20	Unspecified Bug	ntfs_perform_write	4fafd96910add124	✔		
21	Unspecified Bug	btrfs_global_root_insert	509583475828c4fd	✔		
22	Unspecified Bug	jfs_evict_inode	b7455b10da762f2d			
23	Unspecified Bug	erofs_iget	b7455b10da762f2d	✔		
24	Unspecified Bug	do_journal_end	b7455b10da762f2d			
25	Warning	__split_vma	a54df7622717a40d	✔	✔	✔
26	Warning	get_floppy_geometr	9ce08dd7ea24253a			
27	Warning	invalidate_drive	e2f86c02fdc96ca2			
28	Warning	process_fd_request	4d6d7ce9baaf9e67			
29	Warning	udf_truncate_extents	b7455b10da762f2d	✔		
30	Warning	vma_merge	83e5775d7afda68f			
31	Warning	__floppy_read_block_0	83e5775d7afda68f	✔		
32	Warning	btrfs_block_rsv_release	b7455b10da762f2d	✔		
33	Warning	send_hsr_supervision_frame	80bd9028fecadae4			
34	Warning	fd_locked_ioctl	e2f86c02fdc96ca2	✔		
35	Stall	io_ring_exit_work	129af770823407ee			
			Total	**19**	**8**	**4**

To help developers investigate a bug, syzkaller attempts to generate a program that reproduces the bug. But due to the limitations of syzkaller, not all bugs produce a reproducer program. Of our 35 new bugs, syzkaller successfully generated reproducers for 25 of them. Of these 25, 19 successfully triggered the bug. Unfortunately, syzkaller's inability to produce functional reproducers is a known problem. In consulting syzkaller's developers ourselves, we anticipate that kernel nondeterminism is the most likely root cause of our unreproducible crashes. The absence of reproducibility also makes bug reporting and developer-side bug triage more difficult. As the maintainer, Greg Kroah-Hartman points out [80].

> *Reproducer would be great, thanks. Otherwise this goes on the thousands of other "syzbot-found-bugs-with-no-way-to-reproduce" pile that we have...*

We observe, however, that developers can occasionally diagnose and patch bugs without a reproducer. For example, one of the general protection faults we found in `nl802154_trigger_scan` lacked a reproducer, but was confirmed and patched by a kernel maintainer who inferred its cause from the bug report alone [81]. At the time of writing, the remaining bugs are still pending confirmation with developers.

A. The Effects of Configuration Variety

For the 19 bugs that are reproducible, we investigated whether they only apply to the modified configuration file or were applicable to syzkaller's original configuration file (Table VIII). If the bug were only present when little-used options were enabled, it may not be present in typical kernel builds. Of the 19 reproducible bugs, we found that only 4 were specific to the repaired configuration file. The remaining 15 reproduce the bug in kernels built with both the repaired and original configuration file. This indicates that adding configuration variety not only opens the fuzzer to code excluded by its original configuration file, but it also helps alter coverage patterns sufficiently to identify bugs it had not otherwise covered when fuzzing kernels built with the original configuration file.

To further investigate the effects of configuration variety on fuzzing, we traced the coverage of reproducers that triggered the same bug in both the repaired and original configuration files. Our reasoning is that if the same bug is triggered, but follows a different path between the two kernels, then the fuzzer must have taken a path to reach the bug that is not available with the original configuration file, even though the bug is present in the latter's kernel. To track coverage, we ran the reproducers and collected coverage traces with `kcov`, Linux's built-in coverage tracer. Retrieving `kcov` data is only feasible for bugs that do not crash the kernel, since it is the kernel itself maintaining the coverage information. But for 6 of the 7 bugs that do not crash the kernel (they produce a warning instead) we observed differences in the traces between

TABLE VIII: Bugs that depended on configuration variety to be found.

ID	Type	Location	Method
1	Use-After-Free	`hci_conn_hash_flush`	Reproducer
24	Unspecified Bug	`do_journal_end`	Reproducer
29	Warning	`udf_truncate_extends`	Reproducer
3	Use-After-Free	`f2fs_iget`	Reproducer
17	Unspecified Bug	`page_add_anon_rmap`	kcov
8	OOB Access	`fbcon_set_font`	kcov
13	Protection Fault	`efivar_lock`	kcov
34	Warning	`fd_locked_ioctl`	kcov
30	Warning	`vma_merge`	kcov
9	OOB Access	`f2fs_iget`	Call Trace

kernels built with modified and original configuration files, while the remaining 1 appeared unrelated to the repaired configuration file. We quantify the differences between traces as the percentage of differing lines. For the reproducers we traced, these differences ranged from 0.13% to 36%.

For bugs that do crash the kernel, we manually investigated the stack traces reported by syzkaller and to determine whether any functions in the trace were only present in kernels built with repaired configuration files. Our reasoning is that if a function in the stack trace is configuration-specific, then the repaired configuration file was necessary to identify the bug along that specific trace. We found one crash that was indeed configuration specific and detail this bug in the next subsection.

Summary: We identify 10 bugs (Table VIII) whose reachability is *enhanced* by configuration variety—even though these bugs exist in kernels built with syzkaller's original configuration.

B. Case Study: A Configuration-Specific Trace

To better understand how configuration variety helps fuzzers find bugs, we perform a case study of bug #9 from Table VII, which we identified as covering code only available in our modified configuration file but resulting a bug that was reproducible with syzkaller's original configuration file. Its call trace shows the bug location to be function `f2fs_iget` in source file `fs/f2fs/inode.c`, with a previous call occurring to function `f2fs_fill_super`.

Listing 2 displays the relevant parts of the function, including the ultimate call to the buggy function `f2fs_iget`. Caller function `f2fs_fill_super` contains several `#ifdef` conditional compilation directives, which are controlled by the kernel configuration. While relevant features `CONFIG_QUOTA` and `CONFIG_FS_ENCRYPTION` exist in both the modified and configuration configuration file, `CONFIG_FS_VERITY` is only enabled in the original configuration, while our modified configuration file *disables* it, leading to the omission of this source line in the kernel. The `sb` data structure is ultimately passed to the buggy function `f2fs_iget`. Although the stack trace alone does not show direct involvement of the line omitted by `CONFIG_FS_VERITY`, the trace illustrates how modifying the configuration file alters code paths at build time, leading to different coverage patterns and different bugs.

```
1  static int f2fs_fill_super(struct super_block *sb, void *data,
       int silent)
2  {
3  // (code omitted for brevity)
4  #ifdef CONFIG_QUOTA
5    sb->dq_op = &f2fs_quota_operations;
6    sb->s_qcop = &f2fs_quotactl_ops;
7    sb->s_quota_types = QTYPE_MASK_USR | QTYPE_MASK_GRP |
       QTYPE_MASK_PRJ;
8
9    if (f2fs_sb_has_quota_ino(sbi)) {
10     for (i = 0; i < MAXQUOTAS; i++) {
11       if (f2fs_qf_ino(sbi->sb, i))
12         sbi->nquota_files++;
13     }
14   }
15 #endif
16   sb->s_op = &f2fs_sops;
17 #ifdef CONFIG_FS_ENCRYPTION
18   sb->s_cop = &f2fs_cryptops;
19 #endif
20 // krepair disables CONFIG_FS_VERITY
21 #ifdef CONFIG_FS_VERITY
22   sb->s_vop = &f2fs_verityops;
23 #endif
24 // (code omitted for brevity)
25 // call to f2fs_iget containing the bug
26 sbi->node_inode = f2fs_iget(sb, F2FS_NODE_INO(sbi));
```

Listing 2: Abridged code of fs/f2fs/super.c.

VII. Threats to Validity

A. Internal validity

Our replication study produces patch coverage results similar to prior work, suggesting some generality to the limitations of hand-selected patches for continuous testing. We evaluate two popular fuzzers, syzkaller and kAFL, showing that both suffer similar limitations. Our evaluation is for the Linux kernel, which has especially high configurability. While syzkaller supports other kernels, krepair does not.

Fuzzing uses randomization, so multiple runs may exhibit different behavior. As of paper acceptance, there was no known way to replicate individual syzkaller runs, as confirmed by developers [82]. Snapshotting, however, has since been added to syzkaller [83]. While not yet evaluated as of writing, this feature should help with replicability in the future. We mitigated fuzzer randomness in our experiments by comparing the repaired and original configuration against many different patches, which consistently exposed previously-unreported bugs.

While not all bugs are confirmed yet by Linux's developers at the time of writing, we observe that most of our reported bugs have reproducing test cases, proving their existence.

B. External validity

The replication study uses a sample of patches from a single recent year. While that year contains about 17,000 patches and the sample is large enough to have a low margin of error (5%), it is possible that other years or other software would have different patching behavior. A useful future study would be to mine a large number of software repositories and takes large samples of their change histories to evaluate how much configuration variety exists in new code changes.

Our results show that another kernel fuzzer, kAFL, suffers the same configuration testing limitations, but has no publicly-available continuous testing infrastructure. syzkaller is the most popular kernel fuzzer and the only one to our knowledge that has an open source test robot. Additionally, many applications also have high configurability [84], [85], [86], including those using Linux's configuration system, Kconfig.

Our recommendations are specific to continuous fuzzing, since fuzzing has specific needs separate from other testing approaches, such as test suites and static analysis. A similar systematic analysis, however, could be applied to those testing approaches to provide the appropriate recommendations.

Our analysis of configuration selection strategies is based on the current state-of-the-art. Future work could better integrate configuration testing and continuous kernel fuzz testing for further improvements in bug finding, for instance, by encoding multiple configurations in a single binary and fuzzing them together [87], [88], [89] or incorporating configuration selection into existing mutation strategies [90], [11].

VIII. Related Work

A. Configuration Analysis

Research continues to examine the challenges of maintaining and securing configurable software. Melo et al. [91] show developers struggle to perform precise configuration analysis by-hand, particularly for complex code such as OS kernels. Mordahl et al. [92] examine the effectiveness of applying configuration-agnostic static analysis tools to the detection of configuration-dependent application bugs. Ferreira et al. [93] formalize configuration complexity and study its influence on the occurrence of configuration-dependent kernel vulnerabilities. Abal et al. [65] examine previously-fixed configuration-dependent kernel bugs to quantify and understand their configuration complexity. We believe that these and other studies present further opportunities for applying configuration-aware fuzzing, for example, to target pre-identified code regions suspected of containing configuration-dependent bugs.

Prior research is also exploring *static* means of improving configuration-based analysis tasks. C-Reconfigurator [87], Hercules [94], SugarC [89], and Maki [95] demonstrate the feasibility of statically rewriting applications' configuration-dependent code to instead be invokable at runtime; for example, rewriting an `#ifdef`-wrapped code block to instead be guarded by an `if()` statement. Although these techniques have yet to be applied beyond user-space applications, we envision the potential for future synergistic approaches combining configuration-aware kernel fuzzing with additional static kernel analyses and transformations.

Research is also exploring *dynamic* approaches for recognizing differences between program variants. Meinicke et al. [96] introduce the concept of variational traces: a compact representation of the execution path differences among unique variants of the same program. Ferreira et al. [97] perform variational tracing at the call-graph-level. Kaoudis et al. [98] improve the power of variational trace collection at the control- and data-flow level by leveraging LLVM-based compiler instrumentation. We anticipate that advancements in kernel-level execution profiling will facilitate even faster and more effective configuration-aware kernel fuzzing.

Some prior work has explored continuous testing for software product lines [99] albeit not for kernel fuzzing. Other prior work has looked at command-line argument fuzzing for user-space programs [100]. In our work, however, we use existing, unmodified kernel fuzzing algorithms with configuration repair to improve configuration variety.

B. Kernel Fuzzing

Numerous kernel fuzzers have emerged over the years employing different architectures and techniques. By far the most popular is Google's syzkaller [10]—among the first kernel fuzzers to borrow successful principles from application fuzzers like AFL [45], e.g., code coverage guidance, random mutation, and grammars. Intel's kAFL [11] adopts many of the same features, but obtains far greater speed from its faster hypervisor-accelerated VM snapshotting. While the majority of today's kernel fuzzers target open-source kernels such as Linux, kAFL [11] and NTFUZZ [101] are among the few available Windows kernel fuzzers.

Many industrial and academic enhancements are improving kernel fuzzing speed and effectiveness. DIFUZE [49], SyzDescribe [15], and FUZZNG [16] extend the reach of Syzkaller by automating generation of system-call-specific input specifications for its "SyzLang" [10] grammar format. Dr.Fuzz [102], DriFuzz [103], DevFuzz [18], and PrIntFuzz [17] combine static and dynamic analyses to synthesize virtual device models, increasing coverage when fuzzing their associated device drivers. Agamotto [14] and Horus [104] both accelerate kernel fuzzing via incremental process snapshotting and optimized host–VM communication, respectively. As configuration-aware fuzzing is orthogonal to the fuzzer used, we expect that combining it with different fuzzing advancements will yield improved capabilities in finding configuration-dependent kernel bugs.

IX. Conclusion

Fuzzers have been very successful at finding kernel bugs, but our replication study shows that the use of predefined configuration settings leads to missed patches when fuzzers stop to rebuild their kernel, nullifying the benefits of continuous testing. We systematically analyzed the challenges of configuration testing for continuous fuzzers and provided guidelines to configuration and fuzzing researchers, demonstrating how fuzzers can improve both continuous and configuration testing without sacrificing performance. Our evaluation confirms a substantial increase in previously-undiscovered bugs by modifying only the configuration selection strategy while preserving fuzzer performance.

Acknowledgments

We would like to thank to anonymous reviewers for their feedback which has helped improve the paper. This work was supported in part by NSF grant CCF-1941816.

References

[1] "Top 500," 2020, https://www.top500.org/statistics/sublist/.

[2] "W3Techs Surveys: Usage statistics of Unix for websites," https://w3techs.com/technologies/details/os-unix/all/all, 2019.

[3] "Iot developer survey results," https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2018.pdf, 2018.

[4] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, "Evolution of the Linux kernel variability model," in *International Conference on Software Product Lines*. Springer, 2010, pp. 136–150.

[5] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is the linux kernel a software product line," in *Proc. SPLC Workshop on Open Source Software and Product Lines*, 2007.

[6] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[7] IEEE, "IEEE Standard for Configuration Management in Systems and Software Engineering," 2012.

[8] P. Gazzillo, "Inferring and securing software configurations using automated reasoning," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, 2020.

[9] M. Dellago, A. C. Simpson, and D. W. Woods, "Exploit brokers and offensive cyber operations," *The Cyber Defense Review*, 2022.

[10] D. Vyukov, "syzkaller," https://github.com/google/syzkaller, 2023.

[11] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *USENIX Security Symposium*, ser. USENIX, 2017.

[12] K. Kim, T. Kim, E. Warraich, B. Lee, K. R. Butler, A. Bianchi, and D. J. Tian, "Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[13] H. Peng and M. Payer, "{USBFuzz}: A framework for fuzzing {USB} drivers by device emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2559–2575.

[14] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *USENIX Security Symposium*, 2020.

[15] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 3262–3278.

[16] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions." in *Network and Distributed Systems Security Symposium (NDSS)*, 2023.

[17] Z. Ma, B. Zhao, L. Ren, Z. Li, S. Ma, X. Luo, and C. Zhang, "Printfuzz: fuzzing linux drivers via automated virtual device simulation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 404–416.

[18] Y. Wu, T. Zhang, C. Jung, and D. Lee, "Devfuzz: Automatic device model-guided device driver fuzzing," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023.

[19] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel," in *USENIX Security Symposium*, 2022.

[20] J. Corbet, "Some 5.5 kernel development statistics," https://lwn.net/Articles/810639/, 2020.

[21] ——, "Some 5.12 development statistics," https://lwn.net/Articles/853039/, 2021.

[22] ——, "Some 5.19 development statistics," https://lwn.net/Articles/902854/, 2022.

[23] "The linux-next integration testing tree," https://git.kernel.org/pub/scm/linux/kernel/git/next/linux-next.git/, 2024, accessed: 2024-03-21.

[24] J. Spaans, "Linux kernel mailing list," https://lkml.org/, 2024, accessed: 2024-03-21.

[25] "syzbot," https://syzkaller.appspot.com, 2024, accessed: 2024-03-21.

[26] syzbot, "ci-upstream-linux-next-kasan-gce-root," https://syzkaller.appspot.com/upstream/manager/ci-upstream-linux-next-kasan-gce-root, 2024, accessed: 2024-03.

[27] A. Nogikh, "Re: syzkaller use," https://groups.google.com/g/syzkaller/c/kBcVUaF3O40/m/6SJdp0g4AgAJ, 2024, accessed: 2024-03-21.

[28] KAFL, "config.vanilla.virtio," https://github.com/IntelLabs/kafl.targets/blob/master/linux-kernel/config.vanilla.virtio, 2024, accessed: 2024-03.

[29] "syz-kconf in syzkaller on go.dev," https://pkg.go.dev/github.com/google/syzkaller/tools/syz-kconf, 2023, accessed: 2023-12-05.

[30] "syzkaller/dashboard/config/linux/," https://github.com/google/syzkaller/tree/master/dashboard/config/linux, 2022, accessed: 2024-03.

[31] "syzkaller settings," https://github.com/google/syzkaller/blob/master/docs/linux/setup_ubuntu-host_qemu-vm_x86-64-kernel.md, 2023, accessed: 2023-12-05.

[32] KAFL, "Configure and build target kernel," https://intellabs.github.io/kAFL/tutorials/linux/fuzzing_linux_kernel.html#configure-and-build-target-kernel, 2024, accessed: 2024-03-19.

[33] K. Yaghmour, *Building Embedded Linux Systems*. O'Reilly Media, Inc., 2003.

[34] N. F. Yıldıran, J. Oh, J. Lawall, and P. Gazzillo, "Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times," in *Proceedings of the 32nd ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2024, 2024.

[35] M. Acher, H. Martin, J. A. Pereira, A. Blouin, J.-M. Jézéquel, D. E. Khelladi, L. Lesoil, and O. Barais, "Learning very large configuration spaces: What matters for linux kernel sizes," Ph.D. dissertation, Inria Rennes-Bretagne Atlantique, 2019.

[36] K.-C. Tai and Y. Lei, "A test generation strategy for pairwise testing," *IEEE Transactions on Software Engineering*, Jan. 2002.

[37] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, p. 643–654.

[38] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, no. 5, pp. 83–88, Sep. 1996.

[39] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static analysis of variability in system software: The 90,000 #ifdefs issue," in *USENIX Annual Technical Conference*.

[40] "Mainline Linux git repository," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git, 2021.

[41] A. Maguire, "A zoological guide to kernel data structures," https://blogs.oracle.com/linux/post/a-zoological-guide-to-kernel-data-structures, 2021.

[42] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu, "Set the configuration for the heart of the os: On the practicality of operating system kernel debloating," *Proc. ACM Meas. Anal. Comput. Syst.*, may 2020.

[43] P. Michael Larabel, "Linux objtool Improvements Help Reduce RAM Usage & Build Time During Large Kernel Builds," https://www.phoronix.com/news/Linux-objtool-allyesconfig-RAM, 2023.

[44] "CVE-2023-3161," https://access.redhat.com/security/cve/cve-2023-3161.

[45] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *USENIX Workshop on Offensive Technologies*, ser. WOOT, 2020.

[46] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically Generating Pathological Inputs," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA, 2018.

[47] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2829–2846.

[48] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *IEEE Cybersecurity Development Conference*, 2016.

[49] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[50] M. Fleischer, D. Das, P. Bose, W. Bai, K. Lu, M. Payer, C. Kruegel, and G. Vigna, "ACTOR: Action-Guided kernel fuzzing," in *USENIX Security Symposium*, 2023.

[51] "Kconfig language," https://android.googlesource.com/kernel/common/+/refs/heads/android-mainline/Documentation/kbuild/kconfig-language.rst, 2023, accessed: 2023-12-05.

[52] "FreeBSD Handbook: Chapter 10. Configuring the FreeBSD Kernel," https://docs.freebsd.org/en/books/handbook/kernelconfig/, 2023.

[53] "Kconfig language," https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html, 2023, accessed: 2023-12-05.

[54] "The netbsd guide: Tuning netbsd, 19.9.2. configuring the kernel," https://www.netbsd.org/docs/guide/en/chap-tuning.html#tuning-kernel-configure, 2023, accessed: 2023-12-05.

[55] "Openbsd faq: Building the system from source," https://www.openbsd.org/faq/faq5.html, 2023, accessed: 2023-12-05.

[56] "How to build xnu," https://github.com/apple-oss-distributions/xnu#how-to-build-xnu, 2023, accessed: 2023-12-05.

[57] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, may 2023.

[58] N. F. Yıldıran, J. Oh, J. Lawall, and P. Gazzillo, "Artifact from "Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times"," Feb. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10626343

[59] S. Hasanov, S. Nagy, and P. Gazzillo, "Artifact from "A Little Goes a Long Way: Tuning Configuration Selection for Continuous Kernel Fuzzing"," Aug. 2024. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.10854982

[60] "Linux kernel patch submission checklist," https://docs.kernel.org/process/submit-checklist.html?highlight=allnoconfig, 2024.

[61] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018.

[62] X. Tan, Y. Zhang, J. Lu, X. Xiong, Z. Liu, and M. Yang, "Syzdirect: Directed greybox fuzzing for linux kernel," in *ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[63] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "Syzgen: Automated generation of syscall specification of closed-source macos drivers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 749–763.

[64] "Common Vulnerabilities and Exposures Database," https://www.cve.org/, 2024, accessed: 2024-03-21.

[65] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[66] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing," in *ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[67] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: speed up fuzzing with coverage-sensitive tracing and scheduling," in *IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[68] L. Stone, R. Ranjan, S. Nagy, and M. Hicks, "No linux, no problem: Fast and correct windows binary fuzzing via target-embedded snapshotting," in *USENIX Security Symposium*, 2023.

[69] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing New Operating Primitives to Improve Fuzzing Performance," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2017.

[70] S. Nagy and M. Hicks, "Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2019.

[71] "The Linux Kernel Archives," https://kernel.org/, 2024.

[72] https://github.com/paulgazz/kmax, 2024, accessed: 2024-02-06.

[73] "Warning about WARN_ON()," https://lwn.net/Articles/969923/, 2024.

[74] "What to do in response to a kernel warning," https://lwn.net/Articles/876209/, 2021.

[75] "Deprecated Interfaces, Language Features, Attributes, and Conventions," https://docs.kernel.org/process/deprecated.html#bug-and-bug-on, 2024, accessed: 2024-08-15.

[76] "Development tools for the kernel," https://docs.kernel.org/6.6/dev-tools/index.html, 2024.

[77] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *IEEE Symposium on Security and Privacy (SP)*, 2020.

[78] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[79] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing," in *USENIX Security Symposium*, 2021.

[80] G. Kroah-Hartman, "Re: Syzkaller found a bug: KASAN: use-after-free Read in do_update_region," https://lore.kernel.org/lkml/Y35v%2FieA0OrF510w@kroah.com/, 2022, accessed: 2024-03-20.

[81] "general_protection_fault," https://lore.kernel.org/netdev/20230301154450.547716-1-miquel.raynal@bootlin.com/, 2023.

[82] A. Nogikh, "using random seeds of syzkaller," https://groups.google.com/g/syzkaller/c/ABSK8qc9zfw/m/8Bs4MDWxBgAJ, 2022, accessed: 2024-08-15.

[83] D. Vyukov, "add qemu snapshotting mode," https://github.com/google/syzkaller/commit/4d77b9fe7da3d014943a16cb4b9a4ca3a531521a.

[84] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *IEEE Transactions on Software Engineering*, 2013.

[85] N. Wells, "Busybox: A swiss army knife for linux," *Linux Journal*.

[86] R. Fielding and G. Kaiser, "The apache http server project," *IEEE Internet Computing*, vol. 1, no. 4, pp. 88–90, 1997.

[87] A. F. Iosif-Lazar, J. Melo, A. S. Dimovski, C. Brabrand, and A. Wasowski, "Effective analysis of C programs by rewriting variability," *CoRR*, vol. abs/1701.08114, 2017. [Online]. Available: http://arxiv.org/abs/1701.08114

[88] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, p. 907–918.

[89] Z. Patterson, Z. Zhang, B. Pappas, S. Wei, and P. Gazzillo, "Sugarc: Scalable desugaring of real-world preprocessor usage into pure c," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, p. 2056–2067.

[90] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[91] J. Melo, F. B. Narcizo, D. W. Hansen, C. Brabrand, and A. Wasowski, "Variability through the eyes of the programmer," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. IEEE Press, 2017, p. 34–44.

[92] A. Mordahl, J. Oh, U. Koc, S. Wei, and P. Gazzillo, "An empirical study of real-world variability bugs detected by variability-oblivious tools," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, 2019, p. 50–61.

[93] G. Ferreira, M. Malik, C. Kästner, J. Pfeffer, and S. Apel, "Do# ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel," in *Software Product Line Conference*, 2016.

[94] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, and S. Apel, "Variability encoding: From compile-time to load-time variability," *Journal of Logical and Algebraic Methods in Programming*, vol. 85, 07 2015.

[95] B. Pappas and P. Gazzillo, "Semantic analysis of macro usage for portability," in *Proceedings of the 46th International Conference on Software Engineering*, ser. ICSE '24, 2024.

[96] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake, "On essential configuration complexity: Measuring interactions in highly-configurable systems," in *IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[97] G. Ferreira, C. Kästner, J. Pfeffer, and S. Apel, "Characterizing complexity of highly-configurable systems with variational call graphs: Analyzing configuration options interactions complexity in function calls," in *Symposium and Bootcamp on the Science of Security*, 2015.

[98] K. Kaoudis, H. Brodin, and E. Sultanik, "Automatically detecting variability bugs through hybrid control and data flow analysis," in *IEEE Security and Privacy Workshops (SPW)*, 2023.

[99] I. do Carmo Machado, J. D. McGregor, and E. Santana de Almeida, "Strategies for testing products in software product lines," *SIGSOFT Softw. Eng. Notes*, p. 1–8, nov 2012.

[100] A. Lee, I. Ariq, Y. Kim, and M. Kim, "POWER: Program Option-Aware Fuzzer for High Bug Detection Ability," in *2022 IEEE 15th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022.

[101] J. Choi, K. Kim, D. Lee, and S. K. Cha, "Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis," in *IEEE Symposium on Security and Privacy (SP)*, 2021.

[102] W. Zhao, K. Lu, Q. Wu, and Y. Qi, "Semantic-informed driver fuzzing without both the hardware devices and the emulators," in *Network and Distributed Systems Security Symposium (NDSS)*, 2022.

[103] Z. Shen, R. Roongta, and B. Dolan-Gavitt, "Drifuzz: Harvesting bugs in device drivers from golden seeds," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1275–1290.

[104] J. Liu, Y. Shen, Y. Xu, H. Sun, and Y. Jiang, "Horus: Accelerating kernel fuzzing through efficient host-vm memory access procedures," *ACM Trans. Softw. Eng. Methodol.*, nov 2023.