KIT: Testing OS-Level Virtualization for Functional Interference Bugs

Congyu Liu
Purdue University
West Lafayette, Indiana, USA

Sishuai Gong
Purdue University
West Lafayette, Indiana, USA

Pedro Fonseca
Purdue University
West Lafayette, Indiana, USA

ABSTRACT

Container isolation is implemented through OS-level virtualization, such as Linux namespaces. Unfortunately, these mechanisms are extremely challenging to implement correctly and, in practice, suffer from functional interference bugs, which compromise container security. In particular, functional interference bugs allow an attacker to extract information from another container running on the same machine or impact its integrity by modifying kernel resources that are incorrectly isolated. Despite their impact, functional interference bugs in OS-level virtualization have received limited attention in part due to the challenges in detecting them. Instead of causing memory errors or crashes, many functional interference bugs involve hard-to-catch logic errors that silently produce semantically incorrect results.

This paper proposes KIT, a dynamic testing framework that discovers functional interference bugs in OS-level virtualization mechanisms, such as Linux namespaces. The key idea of KIT is to detect inter-container functional interference by comparing the system call traces of a container across two executions, where it runs with and without the preceding execution of another container. To achieve high efficiency and accuracy, KIT includes two critical components: an efficient algorithm to generate test cases that exercise inter-container data flows and a system call trace analysis framework that detects functional interference bugs and clusters bug reports. KIT discovered 9 functional interference bugs in Linux kernel 5.13, of which 6 have been confirmed. All bugs are caused by logic errors, showing that this approach is able to detect hard-to-catch semantic bugs.

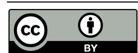
CCS CONCEPTS

• **Security and privacy → Virtualization and security**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

OS-level virtualization, software testing

ACM Reference Format:
Congyu Liu, Sishuai Gong, and Pedro Fonseca. 2023. KIT: Testing OS-Level Virtualization for Functional Interference Bugs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575731



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575731

1 INTRODUCTION

Kernel resource isolation is vital for reliable and secure container isolation. In particular, correct kernel resource isolation must prevent *functional interference* across containers running on the same machine. In practice, Linux containers are implemented using kernel *namespaces*, which prevent a container from accessing (i.e., reading or modifying) resources from other containers, except through authorized means (e.g., valid communication channels). Since functional interference bugs compromise the integrity and confidentiality of containers, they are a major security concern.

Incorrect or insufficient kernel resource isolation can seriously impact container security, especially in multi-tenant environments. In fact, cloud providers are often hesitant to use containers in multi-tenant situations for security reasons [83]. For instance, incorrect isolation might let attackers learn the credentials of another container running on the same machine, which could further lead to cascading attacks on other network-accessible systems [34]. Furthermore, applications critically make assumptions about the services provided by the kernel. Hence, even when incorrect isolation only allows attackers limited resource control, it can enable the exploitation of application bugs that further aggravate the attack impact.

Unfortunately, implementing resource isolation is particularly challenging for kernel developers. This challenge arises from the myriad of kernel resources available (e.g., sockets, files, and timers), which are accessible through the notoriously extensive system call interface [48], and the huge kernel code base. Implementing kernel resource isolation requires adding logical checks, often deep inside the kernel and on each resource access instance, to verify whether the container is allowed to access a resource. This challenge compounds with the complexity of more traditional kernel mechanisms, such as processes, users, and groups. Crucially, a single missed or incorrect check can compromise container security. Thus, it is not surprising that many functional interference bugs have been recently discovered in Linux namespaces, leading to cross-container information leakage [15, 19], denial of service [13, 16, 17], and privilege escalation attacks [14, 21].

Unlike more traditional kernel bugs, such as crashes, functional interference bugs are particularly challenging to detect automatically. In fact, functional interference bugs are often caused by hard-to-catch logic errors [14, 15, 17, 19, 21] that do not cause immediate failures, such as missing results or producing error/warning messages. Thus, traditional kernel fuzzing tools, such as Syzkaller [39], which mainly target bugs involving memory errors [40, 58] (e.g.,

out-of-bound, data race) or obvious failures, cannot comprehensively detect functional interference bugs. Furthermore, while some work [34, 83, 86] has been proposed to address this class of bugs, it is limited to a narrow subset of functional interference bugs.

This paper presents KIT[1], a dynamic testing framework to systematically test OS-level virtualization mechanisms for functional interference bugs. KIT uses *functional interference testing*, a general and principled approach to effectively detect logical errors that compromise container isolation. Functional interference testing conservatively checks whether the kernel enforces strict functional non-interference [37], a strong isolation property, on protected kernel resources.

KIT systematically generates test cases that aim to maximize interference between two programs running in different containers, by issuing carefully crafted system calls and detecting interference across containers. These programs play the roles of a sender and receiver. KIT detects functional interference by executing the *receiver* program twice — once with (①→②→③) and once without (❶→❷→❸) the execution of the *sender* program — and compares the receiver's system call results across both executions to detect divergence (Figure 1).

Crucially, KIT has to address two major challenges to be efficient and practical. First, KIT needs to generate test cases that are likely to trigger functional interference bugs. Considering the large and complex kernel interface, and therefore huge search space in terms of test case generation, this is critical to find bugs efficiently. Second, comparing system call results is non-trivial and must be done without generating false positives that could hinder the frameworks' real-world usability. This requires effectively addressing *legitimate* communication, which allows some functional interference to provide special functionalities [50]. Furthermore, it requires addressing *apparent* interference results, for instance, due to non-determinism, where system call results diverge for extraneous reasons.

KIT addresses the test space size challenge by leveraging the observation that existing functional interference bugs involve a data flow from one container to another over shared kernel variables. Thus, KIT generates effective test case candidates by first analyzing the possible OS-level virtualization-related data flows between two test programs. Specifically, KIT first analyzes the kernel memory accesses of each test program and then generates test cases, composed of two matching test programs that access the same kernel shared variable during profiling. To further increase scalability, KIT clusters test cases that share similar execution behavior and chooses representative test cases from each cluster.

KIT detects functional interference using a partial kernel specification and a deterministic analysis on the system call results. In particular, KIT relies on a partial specification that is incrementally refined by users to simplify result diagnosis. The KIT workflow also uses this specification to further improve test generation. Furthermore, when the system call output divergence is detected, KIT re-runs the receiver programs to identify and analyze the deterministic outcome. To avoid redundant test reports and simplify bug diagnosis, KIT automatically identifies the sender and receiver system call pair that is responsible for the functional interference, then aggregates test reports based on the culprit system call pair.

[1]Kernel isolation tester

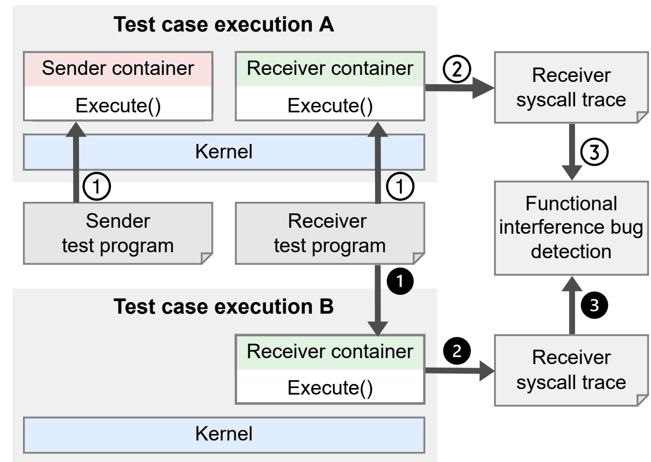


Figure 1: Overview of functional interference testing.

We evaluated KIT by testing the namespace subsystem of the stable Linux kernel 5.13. In total, we found 9 functional interference bugs, of which 7 bugs pose risks of container information leaks and denial of service. Of the bugs we reported, 3 bugs have already been fixed with 2 patches integrated upstream. Our results show that KIT is effective at discovering general functional interference bugs, which are particularly evasive and not the target of existing testing frameworks. Additionally, when using KIT to detect 7 documented severe semantic functional interference bugs in old Linux kernels, KIT detects 5 of them, showing its effectiveness in detecting hard-to-find, exploitable functional interference bugs.

This paper makes the following contributions:

- A general and principled dynamic testing technique to discover functional interference bugs in OS-level virtualization, which (1) leverages profiling-based kernel data flow analysis to generate test cases that encourage inter-container kernel data flow and (2) detects functional interference bugs through a novel test case execution and system call trace analysis approach.
- An approach to prevent false positives by filtering (1) irrelevant system call results based on an input specification and (2) non-deterministic system call results through test re-execution.
- A clustering technique to avoid redundant test reports by aggregating them based on the culprit sender and receiver system call pairs involved.
- The implementation of KIT, a practical dynamic testing framework for the Linux namespace subsystem available at https://github.com/rssys/kit.
- An evaluation of KIT, which found 9 functional interference bugs in the Linux namespace subsystem, of which 3 have been fixed upstream.

2 BACKGROUND AND MOTIVATION

Containers are widely used in data centers for portable and scalable application deployment. Containers are particularly appealing to developers because they provide efficient isolation, allowing safe

resource sharing across applications, high resource utilization, and fast performance. Under the hood, containers require a runtime, such as Docker [60] and LXC [59], and an OS-level virtualization mechanism. While the runtime is responsible for implementing a consistent and convenient execution environment across machines, the OS-level virtualization mechanism is responsible for *isolating containers*. In Linux, container isolation is implemented using kernel namespaces [46].

2.1 Linux Namespaces

Linux namespaces isolate kernel resources per container, such as files and process IDs, allowing containers to have an independent view of the kernel resources. Linux namespaces are designed to be flexible, so they provide applications with some control over which resources are isolated. In particular, there are eight namespaces types, as shown in Table 1, each of which only protects a specific kernel resource type. This allows different container runtimes to choose a specific combination of resources to isolate, according to their requirements.

To isolate a kernel resource type, a process creates and joins a namespace instance with the `unshare` system call, where the process can specify namespace types with flags (e.g., use `CLONE_NEWNET` to specify net namespace). A namespace instance can be assigned to a process or a group of processes. The kernel then ensures that processes in the same namespace group share the same resource view, while external processes cannot access it. For each namespace type, the process is always associated with one namespace instance and can (restrictively) switch between different namespace instances via the `setns` system call. On creation, the new process can join namespace instances of different namespace types if the parent specifies the corresponding namespace flags in the `clone` system call or by inheriting the namespace(s) from the parent.

Namespace bugs. Linux provides many system calls for processes to interact with a wide range of kernel resources, which makes kernel resource isolation particularly challenging to implement. Because resource isolation is a cross-cutting kernel function, many kernel services need to be correctly implemented to ensure effective kernel resource isolation. Hence, to protect such a large interface, developers need to consider every system call that directly or indirectly accesses protected kernel resources. This makes kernel development challenging and error-prone, especially implementing system calls with complex semantics, such as those that use different kernel resource types. For instance, the PID namespace is heavily used to create an isolated process ID space for processes within the PID namespace instance. However, a bug [62] found in Linux v4.17 enabled a process running in one PID namespace instance to access (using an IPC statistic system call) PIDs of processes in another PID namespace instance. The root cause of this bug was that developers neglected the need to add PID namespace isolation to the system call of a seemingly unrelated kernel subsystem, the IPC stack.

2.2 Testing Kernel Resource Isolation

Although several OS testing approaches have been proposed, current approaches do not target general functional interference bugs.

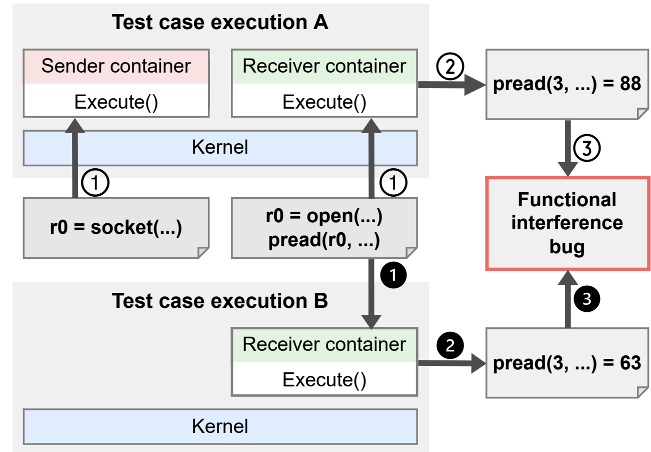


Figure 2: How KIT found bug #1.

In particular, traditional testing approaches that aim to detect memory errors or obvious failures are not effective and efficient at finding functional interference bugs [14, 15, 17, 19, 21].

One prior work [34] aims at finding OS-level virtualization bugs that lead to information leakage. It detects information leakage by comparing the same file in `procfs` and `sysfs` from the host and container. However, it cannot discover information leakage bugs in other kernel interfaces, and only focuses on a tiny subset of the resources protected by namespaces in Linux. Moreover, it does not explore different kernel states. This prevents discovering information leakage bugs that are only triggered in rare kernel states.

Recent work proposes a static analysis approach [83] to discover kernel abstract resources that are vulnerable to container-based denial-of-service attacks. However, static analysis struggles to be sound in large and complex systems, especially when the target system extensively uses pointers, which is the kernel case. Furthermore, this approach is specifically designed to find resource-exhaustion bugs, and it is unclear how to extend it to discover other and more serious types of bugs, such as information leakage bugs. Compared with prior work, KIT proposes a general approach to find general OS-level virtualization bugs of different types.

Case study: ptype information leak. Figure 2 shows a bug found by KIT in the net namespace. In Linux, the `packet_type` data structure is used to forward packets received by certain network devices to upper networking layers. A `packet_type` is registered by several network protocols and by the packet socket, which intercepts raw packets from network devices and is often used to implement userspace network stacks. The kernel maintains the registered `packet_type` data structures of all net namespace instances in global lists. The contents of `packet_type` structures can be fetched through the Linux procfs file `/proc/net/ptype`. This file, as well as the entire `/proc/net` directory, should be isolated by the net namespaces as stated in the net namespace documentation [47]. However, KIT discovered that, when the sender container creates a packet socket (①), the `read` system call trace of `/proc/net/ptype` (②) in receiver container is different (③ ❸) from the `read` system call trace (❷)

when receiver container is running without the sender program execution (❶). We further found that this is because `/proc/net/ptype` enables a container to read the dump of the structures `packet_type` that are registered by packet sockets in other net namespaces. This leaks not only the `packet_type` contents but also its corresponding packet socket liveness.

This bug cannot be found by prior approaches. Triggering this bug requires executing certain system calls, such as those used to create a packet socket, that is not supported by the prior dynamic testing [34] or resource exhaustion detection [83] approaches.

2.3 Functional Interference Detection

Our goal is to find kernel isolation bugs that allow information leaks and integrity attacks—the most serious container bugs. Simply detecting memory errors or obvious kernel failures (e.g., crashes) may detect some of these bugs, but it would leave out the most evasive container bugs—i.e., those that occur because of missing or incorrect logical checks.

Hence, this work explores the idea of using *functional interference* as a detector for container bugs. Instead of looking for internal errors during kernel execution, this approach aims to detect evidence that the kernel produced the *wrong output*. Besides detecting logical errors, this approach has the benefit of simplifying result analysis; it allows developers to reason about the kernel implementation and compare it against the specification (i.e., documentation). However, using functional interference detection as the basis for an effective kernel testing tool involves several challenges.

Challenges. Systematically discovering functional interference bugs is challenging for two reasons. First, efficiently *triggering* functional interference—i.e., catching the kernel red-handed—is difficult because it requires two system call sequences, where one can affect the other when running in two containers. Given the huge and complex kernel interface, such test cases are particularly inefficient to generate through brute-force approaches. Compared with traditional kernel testing approaches (e.g., traditional kernel fuzzers) where only one system call sequence serves as the test case, the search space functional interference testing is quadratic. Second, effectively *detecting* functional interference bugs is challenging because of false positives and non-determinism. Linux namespaces protect many kernel resources, but not all. Thus, functional interferences detected on unprotected kernel resources are not bugs and such results should be filtered out to make result analysis practical.

3 PRACTICAL FUNCTIONAL INTERFERENCE TESTING

This section proposes *functional interference testing*, a method that uses functional interference as a strategy for finding container bugs. It addresses two challenges: efficient test case generation and effective functional interference bug detection.

3.1 Efficient Test Case Generation

KIT's test case generation relies on the key observation that the root cause of functional interference is inter-container communication over the shared kernel memory. In other words, a container (*sender*) can only interfere with another if it modifies a kernel shared memory region that is used to process a request by a process of another container (*receiver*). For example, in the `ptype` information leakage bug discussed in §2.2, the culprit inter-container kernel data flow involves two processes: (1) One process creates a packet socket in its net namespace, causing the `packet_type` shared in the kernel to be updated; (2) another process in another net namespace reads the file `/proc/net/ptype`, causing the kernel to read the shared `packet_type` list. Thus, effective test cases that trigger functional interference must trigger some form of inter-container communication.

Table 1: Linux namespace types. Different namespaces protect different classes of kernel resources.

Namespace type	Kernel resource isolated
PID	Process ID
Mount	Mount point
UTS	Hostname
IPC	System V IPC; POSIX message queue
Net	Network stack
User	UID; GID; capabilities
Cgroups	Cgroups root directory
Time	System time

3.2 Effective Functional Interference Bug Detection

Functional interference testing detects the functional interference from a sender program to a receiver program. This is achieved by analyzing the execution trace of the receiver program when it executes with and without a preceding execution of the sender program. In particular, functional interference testing analyzes the system call traces of the receiver program between two executions. Intuitively, if the preceding execution of the sender program causes functional interference on the receiver program, the receiver will have a different system call trace if it runs without the sender program.

Detecting bugs that trigger functional interference requires a strategy to mitigate false positives caused by two factors. First, functional interference occurs on resources not protected by namespaces (§2.3). Hence, functional interference testing uses an interactive strategy where the user incrementally provides a partial specification for the framework to filter system calls that access resources not protected. Furthermore, it uses this information to inform test case generation. Second, some system call trace divergences across executions are caused by non-determinism. To address non-determinism and avoid false positives, functional interference testing uses a systematic execution environment that executes tests from a stable machine state and reruns the receiver program multiple times to identify and ignore non-deterministic system call results.

4 KIT DESIGN

KIT uses a pipelined architecture (Figure 3) with four stages to test kernels for functional interference bugs. First, KIT generates test cases, which consist of pairs of system call sequences designed to trigger functional interference bugs across containers. Inspired by Snowboard [38], KIT implements a profile-based data flow analysis

and a clustering strategy to generate and distill effective test cases. Second, KIT executes the test cases in two containers to exercise the kernel namespace implementation and traces the system call results. Third, KIT performs a systematic analysis on the system call trace results to help developers accurately identify functional interference bugs. Finally, KIT aggregates test reports caused by the same or similar functional interference so that users can investigate unique functional interference cases efficiently.

4.1 Test Case Generation

KIT takes a set of kernel test programs—sequences of system calls—as input, which can be generated by external tools such as fuzzers. Then KIT generates test cases to trigger functional interference. Each test case conceptually consists of a *sender* and a *receiver* program, which execute in different containers. Intuitively, the sender aims to modify kernel resources that are supposed to be isolated and the receiver aims to detect modifications to those kernel resources.

Finding the right pair of sender and receiver programs that modify and fetch the same namespace-protected kernel resources is crucial for triggering functional interference bugs. However, this is challenging due to the complex kernel interface and quadratic test space (i.e., a pair of programs as opposed to a single program). KIT uses two techniques to improve search effectiveness and efficiency. First, KIT finds test program pairs that are likely to have kernel inter-container data flows, which are necessary for functional interference to happen. It uses a dynamic data flow analysis that profiles the memory accesses trigger by each test program and then finds pairs of programs that trigger write and read accesses to the same memory location. Second, KIT prioritizes test cases that trigger unique kernel behaviors as testing similar behaviors is less rewarding than testing unique ones. KIT uses several heuristics to cluster test cases that may trigger the same kernel behavior, and only executes one test case from each cluster to improve efficiency.

4.1.1 Find Inter-container Communication. To find inter-container data flows that can be triggered by each pair of sender and receiver programs, KIT profiles and analyzes the kernel memory accesses triggered by each test program. If it finds that two test programs separately trigger a write and read memory access to a shared memory region, then it deems that the two programs may have an inter-container data flow.

The kernel behavior, such as the memory access pattern triggered by a test program, largely depends on its execution environment including the container configuration and initial kernel state. Therefore, profiling programs in different and arbitrary execution environments would make it challenging to accurately analyze the test programs. Instead, inspired by other works [30, 38], KIT always uses the same execution environment when profiling each test program. Specifically, it boots the target kernel in a VM and creates two user-level processes. KIT configures the two processes to run in two different namespaces (i.e., containers) and then creates a virtual machine snapshot. This snapshot is always reloaded before KIT profiles a test program.

During the test program execution, KIT relies on kernel instrumentation to collect information about the kernel memory accesses, such as the memory addresses accessed, whether it is a write or read, the address of the instruction that causes the memory access, and the current call stack. To avoid collecting memory accesses that are irrelevant to the test program (e.g., made by background threads), KIT identifies the kernel thread handling system calls made by the test program and only traces memory accesses made by this kernel thread.

Once KIT profiles the execution of every test program, it analyzes the memory accesses to generate functional interference test cases. A pair of test programs that has potential inter-container data flows is promising, but it will only trigger functional interference if the data flow happens over a namespace-protected resource. Thus, KIT only generates a functional interference test case when the read memory access involved in the data flow is caused by a system call that accesses namespace-protected resources (§4.3.1). This is because if the reader is not a system call that accesses namespace-protected resources, then the reader system call cannot be used to detect namespace functional interference bugs. KIT ignores kernel data flows that do not involve namespace-protected resources because exercising them would not be effective at functional interference testing.

4.1.2 Cluster Test Cases. Next, KIT clusters test cases that may trigger similar namespace behavior (e.g., the same functional interference bug) to reduce the testing workload and improve the efficiency of finding functional interference bugs. The main idea is to cluster similar test cases based on the properties of potential inter-container kernel data flows triggered by test cases. If two test cases can cause similar inter-container kernel data flows, they are likely to trigger the same functional interference bug. KIT provides two heuristics as data flow similarity criteria for users to choose: DF-IA and DF-ST.

DF-IA defines data flow that involves the same write and read kernel instructions as similar. DF-ST extends DF-IA in that it also considers the call stacks in which the write and read instructions are executed. In particular, DF-ST only considers two data flows similar if they (1) involve the same write and read instructions and (2) execute the instructions involved under the same call stack context, which is defined as the sequence of function IDs (§5) in the call stack. To avoid cluster explosion, the call stack depth can be limited with a configurable constant.

4.2 Test Case Execution

KIT executes the generated test cases to exercise the potential inter-container data flow and find functional interference. KIT iterates over all test case clusters (§4.1.2) and only chooses one test case to execute from each cluster. KIT uses the VM snapshot to run each test program in a different container so that it can exercise the inter-container data flows and trace the system call results.

KIT executes a test case twice. As shown in Figure 1, in one execution, it first executes the sender program in the sender container, and then executes the receiver program, during which it collects the system call trace of the receiver. In another execution, KIT skips the sender program execution and only executes the receiver program, in which another system call trace is collected. The system call trace contains the execution results of the system calls, including arguments, return value, and error number. The two receiver system call traces are then used to detect functional interference (§4.3).

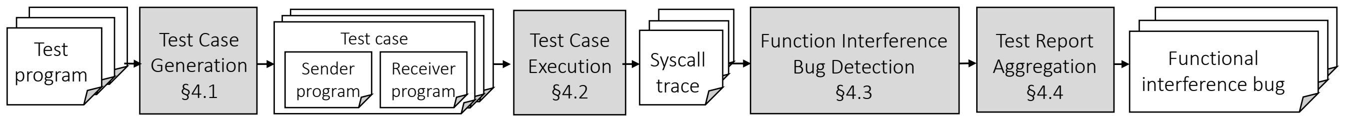


Figure 3: Design overview of KIT.

Algorithm 1 Compare two system call trace abstract syntax trees.

Input: T_a, T_b: Two system call trace abstract syntax tree nodes.
Output: D: List of different tree nodes.

1: **function** SyscallTraceCmp(T_a, T_b)
2: **if** $T_a.det$ **and** $T_b.det$ **then**
3: $la, lb \leftarrow$ Len($T_a.ch$), Len($T_b.ch$)
4: **if** $T_a.val \neq T_b.val$ **or** $la \neq lb$ **then**
5: $D \leftarrow D \cup (T_a, T_b)$
6: **else**
7: **for** $i \leftarrow 0$ to $la - 1$ **do**
8: $D \leftarrow D \cup$ SyscallTraceCmp($T_a.ch[i]$, $T_b.ch[i]$)
9: **return** D

4.3 Functional Interference Bug Detection

KIT compares the system call traces of receiver program when it runs with and without the preceding execution of the sender program. To reduce false positive functional interference, KIT identifies and excludes the system calls results that are non-deterministic or unrelated to namespace-protected resources.

4.3.1 Identify System Call Accessing Protected Resources. KIT analyzes the test program to identify system calls that access namespace protected resources. The identification algorithm relies on a partial input specification, which is provided by the KIT user.

The specification supports two encoding formats. First, users can write callback checker functions to select system calls by comparing call signatures (e.g., call name). Second, users can specify file descriptor types to select system calls that either use or return them. It is efficient to select system calls that access namespace-protected resources that require specific file descriptors as the system call parameter. For instance, when accessing system V message queues [49], the queue ID is generally a system call parameter (e.g., `msgget(id,flag)`). Thus, to test system V message queue, one can provide the queue ID file descriptor as a rule to KIT, which will then select all system calls that either use or return the queue ID. In this way, manually collecting all corresponding system calls that access certain kernel resources and writing callback checker functions to select each of them is no longer necessary.

4.3.2 Identify Non-deterministic Results. Some system calls can produce non-deterministic results, which vary across runs. KIT needs to identify such results in the receiver program so that it does wrongly flag such cases as functional interference. Even worse, some system calls have part of their results that are non-deterministic and parts that are deterministic. For instance, the `fstat` system call produces not only the non-deterministic results, such as timestamps, but also deterministic results, such as the file size. Naively ignoring all results produced by such system calls would prevent KIT from finding important classes of functional interference bugs. Thus, KIT needs a fine-grained trace comparison algorithm that can ignore non-deterministic results during comparison and an automatic approach to identify non-deterministic system call results.

KIT employs a fine-grained system call trace comparison algorithm (Algorithm 1), which compares the abstract syntax trees (AST) of two system call traces and reports the tree differences. Comparing AST differences instead of comparing plain system call trace text enables identifying or ignoring fine-grained system call result differences. A similar approach has also been applied to detect fine-grained source code changes between different versions [28]. To compare two traces, the algorithm recursively traverses two ASTs (lines 6–8), and reports differences when two tree nodes do not match (lines 4–5). Specifically, each node has a `det` flag, which specifies if the system call results represented by the current node and its sub-tree are deterministic. This flag is set to true by default. During the comparison, if one of the two tree nodes contains a `det` flag set to false, the difference between the two nodes is ignored and their sub-tree traversal halts (line 2).

Many non-deterministic system call results are caused by timing. For instance, the output of certain system calls (e.g., timestamp in the `fstat` system call) depends on the system call invocation time and varies across runs. To systematically identify such cases, KIT re-runs the receiver program multiple times with different starting times, so that system call results that are sensitive to timing vary between different executions. KIT then compares all system call trace ASTs, and sets the `det` flag to false for the nodes that vary between different executions. KIT saves this non-determinism information to disk for each test program to reduce the need to rerun the test program in future testing campaigns.

4.4 Test Report Aggregation

An important task for KIT is to aggregate test reports caused by the same type of functional interference such that only the unique ones are examined by users. KIT's core insight to identify similar test reports is that a specific functional interference case can usually only be triggered and detected by a specific sender and receiver system call. Thus, given a set of test reports, KIT first identifies the pair of system calls responsible for the functional interference in each test report and then aggregates test reports based on the system call pair identified, as they are likely due to the same functional interference.

To find the root-cause sender system calls, KIT uses a differential testing approach — for every system call in the sender program, KIT checks whether skipping this sender call during execution will mask the functional interference. Intuitively, a sender system call is responsible for functional interference if the functional interference does not manifest anymore without this sender call.

Algorithm 2 Identify system call pairs that trigger functional interference in a given test report.

Input: P_S: Sender program; P_R: Receiver program; IR: Indices of the system calls in P_R that are interfered by P_S.
Output: S: System call pair list.

```
1:  function DIAGNOSE(P_S, P_R, IR)
2:      S ← φ
3:      for i ← CALLLEN(P_S) −1 to 0 do
4:          P_S ← REMOVECALL(P_S, i)
5:          IR′ ← TESTFUNCI(P_S, P_R)
6:          ΔIR ← IR − IR′
7:          if ΔIR is φ then continue
8:          S ← S ∪ (i, MIN(ΔIR))
9:          IR ← IR − ΔIR
10:         if IR is φ then break
11:     return S
```

Once the sender system call is found, KIT continues to find the interfered receiver system calls that produce different results with and without the sender call. Because of the control and data dependency in the receiver program, multiple interfered calls can be found often. For instance, a sender system call might cause a receiver system call to fail to create a file descriptor, which will further affect the file descriptor value of the following receiver system calls. In this case, KIT only considers the first interfered receiver system call because the functional interference can already be detected by running it after the sender program.

KIT implements Algorithm 2 to identify the system call pairs that trigger functional interference in a given report, which works as follows. It takes three arguments as input: a sender test program (P_S), a receiver test program (P_R), and the indices of P_R system calls interfered by P_S during functional interference testing (IR). It returns a list S containing pairs of sender and receiver system calls where the sender call is responsible for functional interference on the receiver system call. The algorithm removes each sender system call in inverse order (lines 3–4), runs the new test case, and identifies the receiver system calls that are interfered (IR') in the new test case (line 5). As explained above, by comparing IR and IR', the algorithm finds the receiver system calls (ΔIR) interfered by the removed sender system call i (lines 6–8). As illustrated previously, the algorithm will only add the sender system call i and the first receiver system call in ΔIR to S. The algorithm will then remove the receiver system call indices in ΔIR from IR (line 9) since it has found the sender system call that interferes with the calls in ΔIR. If the algorithm has found the culprit sender system call for all interfered receiver system calls in IR (line 10), then it returns S (line 11).

KIT aggregates test reports based on the identified system call pairs that trigger and detect the functional interference. KIT first aggregates test reports by grouping them by the interfered receiver system call (AGG-R). In each AGG-R group, KIT further aggregates test reports by grouping them on the culprit sender system call (AGG-RS) that interferes with this receiver system call. The system call is represented using its name and the file descriptors used by the system call.

5 IMPLEMENTATION

The implementation of KIT is divided into memory tracing and testing components. KIT memory tracing component is implemented with about 200 lines of code in the kernel and 50 lines of code in the compiler. KIT testing component is implemented with about 7400 lines of Go, C/C++, and shell code, excluding the dependent third-party code.

5.1 Test Case Generation

Kernel memory access tracing. KIT profiles kernel memory access using compiler instrumentation. The compiler instrumentation is implemented based on GCC 9.3 and KASAN's GIMPLE [33] pass. Due to GIMPLE's limitations, memory accesses made by inline assembly kernel code are not instrumented automatically. Instead, KIT relies on existing annotations to instrument such kernel code. In particular, KIT leverages the hook functions used by KASAN and KCSAN. In addition, KIT implements a system call for user-space programs to control profiling and collect the profiling data.

To avoid tracing memory accesses irrelevant to test case execution, several implementation choices are made. First, some kernel subsystems are not instrumented since they are less relevant to OS-level virtualization implementation (e.g., scheduler, memory management, tracing hooks, and debugging modules). Second, during run-time, most memory accesses made during interrupt context (e.g., nmi, hardirq, and softirq) are ignored with the help of the kernel's `in_task()` check function, since they do not usually result from the test program's system call and often lead to non-deterministic traces. Lastly, memory accesses to the kernel stack are ignored since the stack is not shared by containers.

KIT also instruments before and after each kernel function callsite. During runtime, the instrumentation produces an execution trace in chronological order. The trace contains entries of three types: function entry, function exit, and memory access, so that KIT can analyze the current call stack for every memory access entry. The function enter-entry also contains a unique function ID, which is assigned to each kernel function during compiler instrumentation. To recover the call stack for each memory access, when processing the traces, KIT maintains a simulated call stack. KIT pushes the function ID into the simulated call stack when it sees the function call enter-entry and pops the simulated call stack when it sees the function call exit-entry. In this way, the call stack of each kernel memory access trace can be obtained by referring to the simulated call stack. Note that this approach assumes that kernel function calls eventually return. Thus, KIT does not instrument functions that do not return exactly once (e.g., functions with GCC `noreturn` attribute).

Test cases generation and clustering. Similar to Snowboard, KIT uses a multi-dimensional map to process the kernel memory accesses made by test programs. The keys of the map include width, read/write flag, memory address, instruction address, and call stack hash. The value of the map is a list of test programs. To generate the map, KIT processes the kernel memory access trace sequence for each test program and updates the map accordingly. For each kernel memory access trace, the call stack hash is generated with the `SHA-1` value of the function ID sequence of the simulated call stack. To generate and cluster test cases, KIT iterates over kernel

memory regions in the map, finds the overlapped kernel memory regions, pairs the test programs to generate test cases, and clusters them based on the specified keys.

5.2 Test Case Execution

Virtual machine. KIT uses QEMU-KVM for test program profiling and test case execution. A test manager on the host interacts with QEMU using the QMP protocol [5] to create and reload a VM snapshot. To avoid introducing non-determinism into the guest OS network stack, which might affect test case execution results, the host test manager communicates with the guest machine through `virtio-serial`.

Test case executor. The KIT test case executor is implemented based on the Syzkaller executor, which interprets the Syzkaller test program and issues system calls. During the VM snapshot creation, KIT first spawns two Syzkaller executors, which set up their containers and block on waiting for test input. After that, KIT takes the VM snapshot. Before executing each test case, KIT resumes from the VM snapshot, and then feeds the test case to the two executors, which then synchronize with each other to execute the sender and receiver test programs in order (i.e., run the sender program first, then run the receiver program).

Container setting. KIT sets up containers to avoid reproducing documented functional interference over protected resources so that it can focus on finding new ones. Hence, KIT tunes a few container settings, which are determined by referring to the documentation or existing container settings. For instance, KIT uses `ulimit` to prevent resource contention on message queues (a resource protected by IPC namespaces) across namespaces, which could cause false positive reports.

System call result decoding library. KIT decodes the system call results to text with a system call decoding library, which we customize from strace [9]. In particular, we customize the strace's internal functions `umoven` and `umovestr`, which are used to copy data from the `ptrace` tracee address space, by changing them to directly copy data from the current process's address space.

Distributed testing. KIT can run distributed tests, so it operates in either the client or server mode. When running in server mode, KIT exposes several RPC services to clients to distribute VM snapshots, transfer test cases, and collect test results.

5.3 Functional Interference Bug Detection

The system call identification is implemented using Syzlang — Syzkaller's system call description framework. KIT allows users to describe a file descriptor using a Syzlang resource identifier, which uniquely represents the file descriptor type (e.g., UNIX socket has a resource identifier `sock_unix`). As Syzlang only assigns unique resource identifiers to a limited of kernel file descriptors, KIT can also select system calls based on user-provided seed system calls. For instance, if the user highlights a seed system call in the program (e.g., `open("/proc/net/*",...)`), KIT will automatically select any system call that has explicit data dependency on the seed system call.

We create a specification that describes the system calls that access resources protected by namespaces. The process is relatively

```
1  static int ptype_seq_show(...) {
2    ...
4    else if (pt->dev == NULL ||      ← Miss ns check
5             dev_net(pt->dev) == seq_file_net(seq)) {
6      if (pt->type == htons(ETH_P_ALL))
7        seq_puts(seq, "ALL ");
8      else
9        seq_printf(seq, "%04x", ntohs(pt->type));
10     ...
11   }
12   ...
13 }
```

Figure 4: Code snippet of bug #1.

simple since most of the kernel resources we tested can be specified by describing the file descriptor type with Syzlang resource identifiers. The system call signature checker function can be easily written since most of them simply check the system call name and require less than 30 lines of code. In total, we wrote 17 system call checker functions and 57 file descriptor types, in roughly 3 person-hours. The resources we selected span across the PID, mount, net, IPC, and user namespaces, and involve the bulk of the namespace system.

6 EVALUATION

Experimental setup. We ran all evaluation experiments on machines with an AMD EPYC 7402P CPU, 128 GB of memory, and Ubuntu 22.04. We generated test cases using a program corpus created by Syzkaller, consisting of 98853 test programs, and we applied KIT to the stable Linux kernel 5.13 release to find new bugs.

6.1 Finding Functional Interference Bugs

In total, KIT found 9 functional interference bugs in Linux 5.13. To save developers' time, we reported 7 bugs, which, to our best knowledge, were not documented. 6 of them were confirmed and 3 were fixed already as of this writing, with 2 patches merged into the mainline kernel. 4 of the bugs found by KIT cause information leakage and 3 others cause denial of service. Both of these classes can affect the security of containers. Given that containers have been massively deployed and thus namespace code is extensively exercised and scrutinized [43], we believe this result demonstrates the effectiveness of KIT.

During our interactions with developers, we found that some bugs were caused by incomplete support for namespaces instead of incorrect checks. For instance, the namespace support for the RDS socket stopped halfway and the consequent incomplete implementation causes bug #3. Bug #6 shares the same property. In the discussion on SCTP namespace support patch, the kernel developer acknowledged that SCTP association ID space "ought to be" per net namespace, but the bug is not fixed due to the high amount of implementation effort.

Case Study: Bug #1. As discussed in §2.2, this bug allows a user to read the dump of the `packet_type` structure in other net namespaces via `/proc/net/ptype`. The sender program, which creates a packet socket, interferes with the `/proc/net/ptype` content in the other container. Our analysis indicates that this bug is due to the mishandling of the `packet_type` structure in the kernel function

Table 2: Linux namespace functional interference bugs found by KIT.

ID	Container S (C_S) action	Container R (C_R) action	C_R syscall trace diff	Resource	Status
1	Create a packet socket	Read /proc/net/ptype	Show the ptype from C_S	ptype	Fixed
2	Create an exclusive flow label	Transmit data with an unregistered flow label	Transmission fails	IPv6 / flow label	Fixed
3	Bind an RDS socket	Bind an RDS socket	Binding fails	RDS / address	Confirmed
4	Create an exclusive flow label	Connect with an unregistered flow label	Connection fails	IPv6 / flow label	Fixed
5	Create a TCP socket	Read /proc/net/sockstat	Counter in file increases	proto / socket	Confirmed
6	Generate a socket cookie	Generate a socket cookie	Cookie changes	socket / cookie	Known
7	Request an association ID	Request an association ID	Association ID changes	SCTP / assoc_id	Known
8	Allocate protocol memory	Read /proc/net/sockstat	Counter in file increases	proto / memory	Confirmed
9	Allocate protocol memory	Read /proc/net/protocols	Counter in file increases	proto / memory	Confirmed

```
1  static inline struct ip6_flowlabel *fl6_sock_lookup(...) {
2    ...
4    if (static_branch_unlikely(&ipv6_flowlabel_exclusive.key))
5      return __fl6_sock_lookup(sk, label) ? : ERR_PTR(-ENOENT);
6    ...
7  }
8  static struct ip6_flowlabel *fl_create(...) {
9    ...
10   if (fl_shared_exclusive(fl) || fl->opt)
11     static_branch_deferred_inc(&ipv6_flowlabel_exclusive);
12   ...
13 }
```

Shared by all ns

Figure 5: Code snippet of bug #2.

ptype_seq_show(): this function does not check the packet socket's net namespace to determine if the packet_type should be displayed or hidden (Figure 4). After KIT found this bug, we submitted a patch to fix this bug, which was merged into the mainline kernel within a week.

This bug allows attackers to infer information about other containers' workloads. Moreover, since an attacker can easily manipulate the content of this file by creating a packet socket, this bug could be used to construct covert channels [53]. It could also be used to fingerprint hosts to co-locate attacker containers and orchestrate power attacks [34].

Surprisingly, we noticed afterward that another patch had been submitted previously for the same function trying to address a similar information leakage bug. Although the developers fixed the case where this file leaks networking device information, another case where it leaks packet socket information was overlooked. This shows the difficulty in correctly implementing and fixing resource isolation: the complex interactions between different networking layers make it hard to reason about the code even for experienced kernel developers. KIT systematically explores kernel execution paths to automatically help developers identify such bugs.

Case Study: Bug #2. The flow label is an essential field in the IPv6 packet header, which is used to represent packet flows at the networking layer. In Linux, the IPv6 protocol stack, including the flow label, is protected by the net namespace [47]. Hence, different net namespaces can use the same flow labels without collisions.

Linux adopts a two-stage flow label management model. When no exclusive flow label (e.g., a flow label exclusively owned by a user) is registered in the kernel, the kernel allows processes to use any flow labels without explicit registration, skipping expensive exclusive flow label collision checks for connections and data transmissions. Once an exclusive flow label is registered in the kernel, the kernel will use a more strict yet expensive flow label management model. A process must register the desired flow label before usage, otherwise, the data transmission and connection will be rejected.

During testing, KIT found a functional interference bug because the flow label management model was not originally implemented with the namespace isolation in mind, i.e., registering an exclusive flow label should only change the flow label management model in its namespace instance, not others. However, this bug allows a sender container to enable the strict and expensive flow label management for all net namespace instances, by registering one exclusive flow label. Thus, a sender container can decrease the performance of other receiver containers that use the IPv6 flow label, such as QUIC [67], a connectionless networking protocol that multiplexes flows [23]. More importantly, this bug breaks the property that each net namespace has its own flow label namespace, where flow collisions across containers are not possible. Hence, developers might implement the container application without handling the strict flow label management model, assuming that the net namespace will isolate the flow label management models between different net namespace instances. In this case, an attacker could cause a denial-of-service in these containers by registering colliding exclusive flow labels. We reported this bug to the kernel developers, who submitted a patch in two days.

The root cause of this bug is that the state of the flow label management model, ipv6_flowlabel_exclusive, is not per net namespace (Figure 5). Note that this variable is implemented with a jump label optimization, where the jump is implemented by code patching instead of making a normal memory access. This optimization prevents our profiling-based data flow analysis from predicting the inter-container data flow over this variable, as it is not instrumented. However, our random test case generation approach found this bug. Resetting the CONFIG_JUMP_LABEL when compiling the kernel will disable this optimization and allows KIT to identify this bug with the data flow analysis. Furthermore, a more comprehensive data flow instrumentation could add support for these cases.

6.2 Detecting Known Isolation Bugs

We evaluated the effectiveness of KIT in detecting known Linux namespace isolation bugs. To collect bugs, we searched through the Linux git commit log and the CVE vulnerability list [20]. We chose silent bugs that are caused by logic errors (i.e., bugs that

Table 3: Known Linux namespace bugs reproduced by functional interference testing.

ID	Container S (C_S) action	Container R (C_R) action	C_R syscall trace diff	Resource	Kernel	NS
A [72]	Change prio using PRIO_USER	Read prio of the current process	Value changes	prio	4.4	pid
B [11]	Create network devices	Listen on kobject uevent	Receive queue uevents	netdev/queue	3.14	net
C [79]	Setup IPVS	Read /proc/net/ip_vs	Read IPVS information from C_S	IPVS	4.15	net
D [19]	Set nf_conntrack_max	Read nf_conntrack_max	Value changes	nf_conntrack	5.13	net
E [18]	(Host) Create files in /tmp	Read unmounted /tmp via io_uring	Observe newly created files	mount	5.6	mnt

Table 4: Evaluation of different test case generation and clustering strategies. KIT skips redundant test cases across clusters.

Gen	Test cases (M)	Effectiveness
DF-IA	1.13	9/9
DF-ST-1	3.32	9/9
DF-ST-2	6.61	9/9
RAND	8.66	5/9
DF	234.63	

do not crash or hang the kernel), which are the hardest-to-find by users and the main focus of KIT. In fact, all these bugs were found manually instead of by an automated testing tool. All bugs analyzed are recent bugs with reports that include the reproduction steps, so that we can write the test case in C.

In total, we collected 7 known bugs, and KIT was able to reproduce 5 of them (Table 3), showing its effectiveness in detecting severe functional interference bugs. The reproduced bugs were found in different namespaces, including the net, mount, and PID namespaces. Moreover, some of them have been shown to introduce security vulnerabilities. For instance, bug D allows directly changing the global `nf_conntrack` sysctl parameter from any net namespace created by privileged users, which can cause a denial of service; while bug E allows a user in a mount namespace to escape to the host mount points. Both bugs D and E have assigned security advisory reports (CVEs).

Additionally, we found that 2 known Linux namespace isolation bugs can not be detected by functional interference testing. For instance, one bug [85] causes functional interference over a kernel resource that has non-deterministic system call results even without any functional interference, so this bug is ignored by KIT. Another bug [81] requires the receiver test program to know the exact resource ID created by the sender program during runtime, which is not supported by our functional interference testing approach. Although it would be desirable to support such bugs, exploiting them is typically more difficult because attackers cannot deterministically retrieve information from the receiver in one shot, so they are typically less serious.

6.3 Test Case Generation

We further analyzed the effectiveness of different test case generation approaches. We define effectiveness as the ability to discover new functional interference bugs. To see how the inter-container data flow analysis improves test case generation, we implement a

Table 5: Test report filtering effectiveness. "After non-det + resource filtering" represents the final number of filtered reports before aggregation.

	Number	Percentage
Tests executed	1,132,761	
Initial reports	15,353	100%
After non-det filtering	891	5.80%
After non-det + resource filtering	808	5.26%

Table 6: Test report aggregation results. Results include false positives (FP) and cases still under investigation (UI).

	Bug ID									FP	UI	Total
	1	2	3	4	5	6	7	8	9			
Filtered reports	12	22	7	4	3	2	679	2	5	61	11	808
AGG-RS groups	7	12	1	3	1	2	13	1	2	19	10	71
AGG-R groups	5	7	1	2	1	2	2	1	1	4	6	32

random test case generation (RAND) as a baseline approach for comparison. This approach randomly chooses the sender and receiver program from the input corpus to build one test case. Furthermore, we compared two test case clustering strategies that are applied upon inter-container data flows to improve test case effectiveness, namely the instruction address strategy (DF-IA), and the call stack strategy (DF-ST). We evaluated DF-ST with the call stack depth set to one (DF-ST-1) and two (DF-ST-2). When evaluating each clustering strategy, we executed enough test cases so that every cluster was exercised, i.e., at least one of its test cases was executed.

As shown in Table 4, DF-IA, DF-ST-1, and DF-ST-2 are equally effective, as they can detect all new functional interference bugs after exercising all clusters. Furthermore, they significantly distill the number of test cases generated based on data flow analysis (DF). However, RAND is much less effective as it only discovers bugs #1, #2, #5, #7, and #9. This shows the effectiveness of KIT data flow based test case generation.

6.4 Distilling Test Reports

To understand how KIT helps users efficiently diagnose bug reports, we evaluated how KIT filters out false positive reports and then aggregates reports by the same functional interference. The bug reports used in this section were gathered from the DF-IA test case generation strategy, which is representative of the other strategies as well.

Filter test reports. Table 5 shows how KIT filtered the false positive test reports with non-deterministic result identification (§4.3.2)

and by identifying system calls that access protected resources (§4.3.1). Recall that KIT first filters non-deterministic candidate reports where the functional interference is not reproducible, and then it filters reports where the functional interference happens over resources not protected by namespaces.

In total, the two filtering methods removed 14,545 false positive reports from 15,353 test report candidates, showing KIT can significantly reduce the manual efforts for investigation by identifying false positive reports automatically. The non-deterministic filter was more effective than the protected resource filter. We believe that this is due to KIT's test case generation algorithm (§4.1.2). It ensures that the receiver program always contains at least one system call that accesses resources protected by namespaces. Thus, fewer system calls that do not access protected resources are exercised, which leads to fewer false positives.

Aggregate test reports. Recall that KIT uses two report aggregation strategies to simplify analysis: AGG-R, which aggregates reports with the same receiver system call, and AGG-RS, which aggregates reports with the same sender and receiver system call pair. Table 6 includes the number of AGG-R and AGG-RS groups that contain the test cases triggering the functional interference for each bug found. We also counted the number AGG-R and AGG-R groups that contain false positives (FP) reports or reports under investigation (UI). As shown in Table 6, the total number of AGG-R and AGG-RS groups is much smaller than the total number of test reports, and most bugs only involved a couple of clusters, which significantly simplifies the analysis.

Test report aggregation greatly reduced the analysis time of redundant test reports. In total, we spent around 30 person-hours on diagnosing the reports. In particular, we spent approximately 20 person-hours on diagnosing test reports of functional interference bugs in Table 2 and 10 person-hours on the remaining test reports. Diagnosing a test report is inevitably time-consuming, as it involves correctly understanding the namespace implementation, diagnosing the root causes, analyzing past kernel mailing and commit history, and writing patches. As an AGG-RS group aggregates test reports that trigger the same functional interference, we only need to examine one test report from an AGG-RS group. For instance, KIT generated 684 test reports that triggered bug #7 but it was able to aggregate similar reports together and only output 13 AGG-RS groups, therefore we only need to analyze 13 test reports.

False positives. We identified 4 AGG-R and 19 AGG-RS groups as false positives (61 test reports). We observed that all these cases were caused by incomplete test report filtering, where tested resources are not protected by namespaces. For instance, 11 AGG-RS (46 test reports) groups involve functional interference in the minor device number of `procfs`, `ramfs`, and others, which is returned by the `stat` and `fstat` system calls. These are false positives because the minor device number is not protected by namespaces.

Handling false positive test reports with report aggregation is relatively easy. Once the user confirms one false positive test report, the entire AGG-RS group it belongs to can be dropped to avoid other similar false positive reports. Users can even drop the entire AGG-R group to exclude all test reports where the functional interference happens on the same receiver system call. For instance, we dropped an AGG-R group containing 14 test reports, where the interfered receiver system call reads `/proc/crypto`, which is not protected by namespaces.

6.5 Performance

Test case generation. KIT executes each test program four times to get the system call trace and kernel execution trace when running it in the sender and receiver containers. KIT executes each test program twice in both the sender and receiver container. In one execution KIT collects the system call trace and in another execution it collects the execution trace, which includes information about call stacks and kernel memory accesses of the kernel thread. Two trace collections have to run separately as collecting execution traces using instrumentation may affect the system call trace. For instance, executing the instrumentation code slows down the performance and may cause a timeout error for some system calls.

KIT takes less than 9 hours to profile the entire corpus on a single server. This is significantly faster than other approaches to profile memory accesses. For instance, Snowboard takes 80 hours to profile 129,876 test programs with 10 machines running in parallel [38]. Two factors contribute to the higher profiling performance: (1) KIT leverages compiler instrumentation to collect memory access traces and efficiently profiles test program by taking advantage of hardware virtualization (e.g., running with KVM enabled) instead of collecting memory accesses through software emulation; (2) Instead of profiling every kernel memory access, KIT avoids instrumenting the kernel memory accesses that are irrelevant to namespace isolation. KIT analyzes memory traces and generates test cases in a single machine within 30 minutes.

Test case execution. By spawning 110 VMs in total, across 4 servers, this test setting allows KIT to achieve 31.3 test case executions per second. In total, KIT executes 1.13M test cases within 10 hours. The performance of KIT could further benefit from existing fast snapshot mechanisms, such as on-demand-fork [87] and others [68, 70].

7 DISCUSSION

Initial test programs. KIT relies on external kernel testing tools (e.g., Syzkaller) to generate the initial kernel test programs. Undoubtedly, the quality of these test programs is crucial for finding functional interference bugs. For instance, if triggering a certain functional interference bug requires a data flow over a kernel shared variable, but all initial test programs either do not write to or write the original value (i.e., a write that does not change the state) to this kernel shared variable, then KIT cannot detect this functional interference bug. Hence, we expect that KIT's effectiveness will improve with future advances in the thriving field of feedback fuzzers.

Applications to other isolation mechanisms. Although we focus on container bugs due to their popularity and security impact, KIT could be applied to test other isolation mechanisms, such as hypervisor and TEE-based approaches [1, 8, 31, 42, 71, 88]. By design, KIT is able to detect functional interference bugs for the majority types of Linux namespace functional interference bugs, even though all bugs found by KIT are in the network namespace,

possibly due to the complexity of this particular subsystem and the focus of Syzkaller test program generation.

Bug detection. Functional interference testing ignores non-deterministic resources during trace divergence analysis. One drawback of this design is that KIT cannot efficiently test the time namespace since the protected resources (e.g., systems clocks) are non-deterministic. A possible solution is to learn the valid bounds of resource values, caused by non-determinism, through dynamic profiling and detecting inter-container resource interference by identifying bound violations. A similar approach has been formalized in prior work [10], which detects timing side channels. Plus, the current implementation does not support detecting bugs that only expose under complex kernel thread interleavings. However, our study on known functional interference bugs indicates that most bugs can be exposed or even exploited without concurrency. In addition, KIT can be combined with concurrency testing tools [2, 29, 30, 38, 44, 82] to detect concurrency functional interference bugs. We leave this as future work.

False positives. KIT filters system calls that do not access protected resources to reduce false positives. Nevertheless, given the complex system call interface, some system calls that access protected resources may contain states of the resources that are not protected, which could result in false positives. With the test report aggregation (§4.4), KIT reduces the users' effort spent on analyzing redundant false positive functional interference bug reports. Furthermore, we believe a more detailed document of kernel resource isolation could better assist users to identify and diagnose these cases. Additionally, KIT can reduce the non-deterministic test reports with the help of prior works in deterministic execution [22, 57, 64].

8 RELATED WORK

OS-level virtualization testing. Dynamic testing approaches [7, 30, 31, 36, 38, 39, 41, 45, 51, 66, 69, 76] have been proposed to discover general crashes, memory bugs, concurrency bugs, and hypervisor bugs in kernels. However, few works target functional interference bugs in OS-level virtualization implementation. Kernel regression testing [3, 24, 52] relies on test cases written by kernel developers to test OS-level virtualization, but these test cases mainly focus on exercising well-known patterns of functional interference bugs rather than finding new ones. There are a few prior works that look at testing OS-level virtualization. For instance, a recent work [83] proposes a static analysis framework to discover resource exhaustion bugs in OS-level virtualization. Another work [34] aims at discovering information leakage in Linux containers. Pex [86] uses static analysis to identify permission check errors in the Linux user namespace. Nevertheless, none of them targets general resource isolation bugs as KIT does for Linux namespaces.

OS-level virtualization security. One line of work in OS-level virtualization security focuses on the security of container runtime and orchestration toolchains [6, 12, 74, 78, 84]. For instance, one work [78] studies the security implications of using container images in the production cloud. KIT is more related to another line of work that focuses on the security of resource isolation and access control mechanisms provided by the kernel. For example, Gao et al. [35] explore using out-of-band workloads to escape the control group resource limit. Lin et al. [55] study existing attacks against Linux security mechanisms and proposed defense solutions. CNTR [80] reduces the container attack surface by reducing the container image size without compromising functionality. Baston [63] hardens Linux container network stacks via restricted visibility and network traffic isolation. Sun et al. [77] propose the security namespace to enable autonomous security policy configurations for containers. SCONE [4] is a container with Intel SGX support to encrypt I/O data. X-Containers [73] leverages the exokernel and libOS [25] to enforce the inter-container resource isolation with small attack surfaces.

Kernel data flow analysis. Identifying possible container interference requires data flow analysis on the kernel. Recent work in this domain can be divided into two categories. One category, such as Razzer [44], uses static analysis (e.g., points-to analysis) to identify potential data races. By contrast, Krace [82] and Snowboard [38] rely on dynamic executions. Krace executes many random kernel test inputs and monitors data flows by instrumenting every shared kernel memory access. Snowboard takes a set of system call sequences as inputs and dynamically profiles the shared memory access triggered by each sequence. Then it identifies overlapped shared memory accesses between two sequence profiles as possible data flows. KIT adopts a data flow analysis framework that is similar to Snowboard, but introduces new test case clustering strategies and leverages compiler instrumentation to efficiently profile kernel memory accesses.

Non-interference. Non-interference has been used in OS verification [27, 54, 61, 75] and model checking [26] to prove information flow security for critical kernel subsystems. Li et al. [54] prove non-interference for a retrofitted Linux KVM hypervisor that ensures the confidentiality and integrity of VM data. Although verification can provide strong correctness guarantees, it struggles to scale to large and complex systems, such as Linux, and still relies on assumptions that need to be tested [32]. Hence, some prior testing frameworks [10, 65] check non-interference for bug detection, but mainly focus on discovering side-channel vulnerabilities in user-space software. KIT focuses instead on testing the isolation of OS-level virtualization.

9 CONCLUSION

This work introduces KIT, a framework to find functional interference bugs in OS-level virtualization. KIT uses a general method, functional interference testing, that tackles two major challenges in finding functional interference bugs. First, to generate effective test cases that can trigger functional interference, KIT uses a dynamic data flow analysis to identify possible inter-container data flows in the test case and prioritize test cases that exercise unique and untested functional interference. Second, KIT automatically identifies classes of false-positive functional interference that are caused by kernel non-determinism and namespace-irrelevant kernel resources so that KIT can accurately report functional interference bugs. KIT has found 9 new functional interference bugs and many of them have serious impact on container security. Additionally, KIT

can detect many known severe functional interference bugs, showing its effectiveness in detecting vulnerable functional interference bugs.

ACKNOWLEDGMENTS

This work was funded in part by NSF under grants CNS-2140305 and CNS-2145888 and Google. We thank all anonymous reviewers for their feedback, which greatly improved the paper. We also thank Adil Ahmad for his helpful suggestions on the design and evaluation.

A ARTIFACT APPENDIX

A.1 Abstract

KIT is a dynamic testing tool to systematically discover functional interference bugs in OS-level virtualization. Currently, the KIT artifact supports testing Linux namespaces.

A.2 Artifact check-list (meta-information)

- **Program:** kit-artifact
- **Compilation:** The required compilers include gcc, g++, go, and a customized gcc. They can be installed via the script provided.
- **Data set:** A test program corpus generated by Syzkaller. It can be downloaded via the script provided.
- **Run-time environment:** Linux systems; root access required.
- **Hardware:** x86-64 CPU; 128GB memory
- **Output:** Bug reports.
- **Experiments:** Find the new functional interference bugs; Reproduce the known functional interference bugs.
- **How much disk space required (approximately)?:** 256GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 1 day
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GPL-3.0 license.
- **Data licenses (if publicly available)?:** GPL-3.0 license.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7240401

A.3 Description

A.3.1 How to access. The source code of this artifact is available at Zenodo [56] and GitHub: https://github.com/rssys/kit-artifact

A.3.2 Hardware dependencies. The artifact evaluation requires a machine with x86-64 CPU, 128GB memory, and 256GB storage.

A.3.3 Software dependencies. The artifact evaluation requires Linux systems with QEMU and KVM support and root access. Some optional experiments require Docker.

A.3.4 Data sets. The artifact will generate the test cases using a test program corpus generated by Syzkaller as input. The dataset can be downloaded via the script provided in the artifact.

A.4 Installation

Please see the `README.md` file in https://github.com/rssys/kit-artifact. First, install the dependencies required to build KIT. Next, follow the instructions to run a script, which will (1) install the go compiler; (2) patch the Syzkaller code used by the artifact; (3) build the whole artifact, which includes the main testing framework, Syzkaller, a system call trace decoding library, and a customized gcc compiler. Then, prepare the test input required for the artifact evaluation. Run a script to set up the environment to find new functional interference bugs, which will (1) build the instrumented Linux kernel; (2) build a VM image; (3) download a Syzkaller test program corpus. Run another script to set up the environment to reproduce known functional interference bugs, which will download the prebuilt old Linux kernel and VM images to test.

A.5 Evaluation and expected results

The artifact evaluation will cover the following aspects that serve as the key results of this paper: (1) the discovery of 9 functional interference bugs with DF-IA test case generation strategy (Table 2, Table 4); (2) the effectiveness evaluation of test report filtering (Table 5); (3) the statistics on test report aggregation (Table 6); (4) reproducing known functional interference bugs (Table 3). The artifact provides several scripts to automatically run the whole pipeline and reproduce the results. For more details regarding the evaluation, please read the `README.md` file in https://github.com/rssys/kit-artifact. Due to differences in test settings or randomness, there might be slight differences between the results in the paper and those from the artifact evaluation.

REFERENCES

[1] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. 2021. CHANCEL: Efficient Multi-client Isolation Under Adversarial Programs. In *28th Annual Network and Distributed System Security Symposium (NDSS '21)*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/chancel-efficient-multi-client-isolation-under-adversarial-programs/

[2] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. 2021. Kard: Lightweight Data Race Detection with per-Thread Memory Protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)* (Virtual, USA). Association for Computing Machinery, New York, NY, USA, 647–660. https://doi.org/10.1145/3445814.3446727

[3] Linux Kernel Archives. 2022. Linux Kernel Selftests. https://www.kernel.org/doc/Documentation/kselftest.txt.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[5] Fabrice Bellard. 2022. Documentation/QMP. https://wiki.qemu.org/Documentation/QMP.

[6] Thanh Bui. 2015. Analysis of docker security. *arXiv preprint arXiv:1501.02967* (2015).

[7] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2022. Morphuzz: Bending (Input) Space to Fuzz Virtual Devices. In *31st USENIX Security Symposium (Security '22)*. USENIX Association, Boston, MA, 1221–1238. https://www.usenix.org/conference/usenixsecurity22/presentation/bulekov

[8] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *2020 IEEE Symposium on Security and Privacy (S&P '20)*. 1416–1432. https://doi.org/10.1109/SP40000.2020.00061

[9] Vitaly Chaykovsky. 2022. strace. https://strace.io.

[10] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities Using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)* (Dallas, Texas, USA). Association for Computing Machinery, 875–890. https://doi.org/10.1145/3133956.3134058

[11] Weilong Chen. 2014. net: fix "queues" uevent between network namespaces. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=82ef3d5d5f3ffd757c960693c4fe7a0051211849.

[12] Theo Combe, Antony Martin, and Roberto Di Pietro. 2016. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing* 3, 5 (2016), 54–62.

https://doi.org/10.1109/MCC.2016.100
[13] The MITRE Corporation. 2018. CVE-2018-14646. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-14646.
[14] The MITRE Corporation. 2018. CVE-2018-18955. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-18955.
[15] The MITRE Corporation. 2018. CVE-2018-6559. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6559.
[16] The MITRE Corporation. 2019. CVE-2019-11815. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11815.
[17] The MITRE Corporation. 2019. CVE-2019-20794. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-20794.
[18] The MITRE Corporation. 2020. CVE-2020-29373. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-29373.
[19] The MITRE Corporation. 2021. CVE-2021-38209. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-38209.
[20] The MITRE Corporation. 2022. CVE - CVE. https://cve.mitre.org.
[21] The MITRE Corporation. 2022. CVE-2022-0492. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0492.
[22] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)* (Farminton, Pennsylvania). Association for Computing Machinery, New York, NY, USA, 388–405. https://doi.org/10.1145/2517349.2522735
[23] Willem de Bruijn. 2019. ipv6: elide flowlabel check if no exclusive leases exist. https://lists.openwall.net/netdev/2019/07/07/50.
[24] LTP developers. 2022. LTP - Linux Test Project. http://linux-test-project.github.io.
[25] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)* (Copper Mountain, Colorado, USA). Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076
[26] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. 2010. Idle Port Scanning and Non-interference Analysis of Network Protocol Stacks Using Model Checking. In *19th USENIX Security Symposium (Security '10)*. USENIX Association, Washington, DC. https://www.usenix.org/conference/usenixsecurity10/idle-port-scanning-and-non-interference-analysis-network-protocol-stacks
[27] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)* (Shanghai, China). Association for Computing Machinery, New York, NY, USA, 287–305. https://doi.org/10.1145/3132747.3132782
[28] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743. https://doi.org/10.1109/TSE.2007.70731
[29] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding Complex Concurrency Bugs in Large Multi-Threaded Applications. In *Proceedings of the Sixth Conference on Computer Systems (Eurosys' 11)* (Salzburg, Austria). Association for Computing Machinery, New York, NY, USA, 215–228. https://doi.org/10.1145/1966445.1966465
[30] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Broomfield, CO, 415–431. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/fonseca
[31] Pedro Fonseca, Xi Wang, and Arvind Krishnamurthy. 2018. MultiNyx: A Multi-Level Abstraction Framework for Systematic Analysis of Hypervisors. In *Proceedings of the Thirteenth EuroSys Conference (Eurosys '18)* (Porto, Portugal). Association for Computing Machinery, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/3190508.3190529
[32] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth EuroSys Conference (Eurosys '17)*. Belgrade, Serbia. https://dl.acm.org/doi/10.1145/3064176.3064183
[33] Inc. Free Software Foundation. 2022. GIMPLE (GNU Compiler Collection (GCC) Internals). https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.
[34] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. 2017. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '17)*. 237–248. https://doi.org/10.1109/DSN.2017.49
[35] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. 2019. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)* (London, United Kingdom). Association for Computing Machinery, New York, NY, USA, 1073–1086. https://doi.org/10.1145/3319535.3354227
[36] Xinyang Ge, Ben Niu, Robert Brotzman, Yaohui Chen, HyungSeok Han, Patrice Godefroid, and Weidong Cui. 2021. HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)* (Virtual Event, Republic of Korea). Association for Computing Machinery, New York, NY, USA, 366–378. https://doi.org/10.1145/3460120.3484748
[37] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. https://doi.org/10.1109/SP.1982.10014
[38] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-Thread Communication Analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)* (Virtual Event, Germany). Association for Computing Machinery, New York, NY, USA, 66–83. https://doi.org/10.1145/3477132.3483549
[39] Google. 2022. google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.
[40] Google. 2022. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html.
[41] Jesse Hertz. 2022. TriforceLinuxSyscallFuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.
[42] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Savannah, GA, 533–549. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt
[43] Docker Inc. 2022. Docker security | Docker Documentation. https://docs.docker.com/engine/security/#kernel-namespaces.
[44] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (S&P '19)* (San Francisco, CA). 754–768. https://doi.org/10.1109/SP.2019.00017
[45] Dave Jones. 2022. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity.
[46] Michael Kerrisk. 2022. namespaces(7) — Linux manual page. https://man7.org/linux/man-pages/man7/namespaces.7.html.
[47] Michael Kerrisk. 2022. network_namespaces(7) — Linux manual page. https://man7.org/linux/man-pages/man7/network_namespaces.7.html.
[48] Michael Kerrisk. 2022. syscalls(2) — Linux manual page. https://man7.org/linux/man-pages/man2/syscalls.2.html.
[49] Michael Kerrisk. 2022. sysvipc(7) — Linux manual page. https://man7.org/linux/man-pages/man7/sysvipc.7.html.
[50] Michael Kerrisk. 2022. veth(4) — Linux manual page. https://man7.org/linux/man-pages/man4/veth.4.html.
[51] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *27th Annual Network and Distributed System Security Symposium (NDSS '20)* (San Diego, California, USA). The Internet Society. https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/
[52] KUnit. 2022. KUnit - Unit Testing for the Linux Kernel. https://kunit.dev/third_party/kernel/docs/.
[53] Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (oct 1973), 613–615. https://doi.org/10.1145/362375.362389
[54] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (S&P '21)* (San Francisco, CA). 1782–1799. https://doi.org/10.1109/SP40001.2021.00049
[55] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)* (San Juan, PR, USA). Association for Computing Machinery, New York, NY, USA, 418–429. https://doi.org/10.1145/3274694.3274720
[56] Congyu Liu, Sishuai Gong, and Pedro Foseca. [n. d.]. Artifact of KIT: Testing OS-Level Virtualization for Functional Interference Bugs. https://doi.org/10.5281/zenodo.7240401
[57] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)* (Cascais, Portugal). Association for Computing Machinery, New York, NY, USA, 327–336. https://doi.org/10.1145/2043556.2043587
[58] LOCKDEP 2006. ANNOUNCE: Lock validator. http://lwn.net/Articles/185605/.
[59] Canonical Ltd. 2022. Linux Containers. https://linuxcontainers.org/.
[60] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal* 2014 (2014), 2.
[61] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *2013 IEEE Symposium on Security and Privacy (S&P '13)* (San Francisco, CA). 415–429. https://doi.org/

10.1109/SP.2013.35

[62] Nagarathnam Muthusamy. [n. d.]. ipc/msg: Fix msgctl(..., IPC_STAT, ...) between pid namespaces. https://github.com/torvalds/linux/commit/39a4940eaa185910bb802ca9829c12268fd2c855.

[63] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. 2020. BASTION: A Security Enforcement Network Stack for Container Networks. In *2020 USENIX Annual Technical Conference (ATC '20)*. USENIX Association, 81–95. https://www.usenix.org/conference/atc20/presentation/nam

[64] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. Reproducible Containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)* (Lausanne, Switzerland). Association for Computing Machinery, New York, NY, USA, 167–182. https://doi.org/10.1145/3373376.3378519

[65] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. 2019. DifFuzz: Differential Fuzzing for Side-Channel Analysis. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)* (Montreal, Quebec, Canada). IEEE Press, 176–187. https://doi.org/10.1109/ICSE.2019.00034

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

[67] The Chromium Projects. 2022. QUIC, a multiplexed transport over UDP. https://www.chromium.org/quic/.

[68] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (Security '21)*. USENIX Association, 2597–2614. https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[69] 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 (Security '17)*. USENIX Association, Vancouver, BC, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[70] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)* (Rennes, France). Association for Computing Machinery, New York, NY, USA, 166–180. https://doi.org/10.1145/3492321.3519591

[71] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy (S&P '15)*. 38–54. https://doi.org/10.1109/SP.2015.10

[72] Ben Segall. 2022. pidns: fix set/getpriority and ioprio_set/get in PRIO_USER mode. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8639b46139b0e4ea3b1ab1c274e410ee327f1d89.

[73] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)* (Providence, RI, USA). Association for Computing Machinery, New York, NY, USA, 121–135. https://doi.org/10.1145/3297858.3304016

[74] Rui Shu, Xiaohui Gu, and William Enck. 2017. A Study of Security Vulnerabilities on Docker Hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY '17)* (Scottsdale, Arizona, USA). Association for Computing Machinery, New York, NY, USA, 269–280. https://doi.org/10.1145/3029806.3029832

[75] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Carlsbad, CA, 287–305. http://www.usenix.org/conference/osdi18/presentation/sigurbjarnarson

[76] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)* (Virtual Event, Germany). Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3477132.3483547

[77] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *27th USENIX Security Symposium (Security '18)*. USENIX Association, Baltimore, MD, 1423–1439. https://www.usenix.org/conference/usenixsecurity18/presentation/sun

[78] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. 2017. Understanding Security Implications of Using Containers in the Cloud. In *2017 USENIX Annual Technical Conference (ATC '17)*. USENIX Association, Santa Clara, CA, 313–319. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tak

[79] KUWAZAWA Takuya. 2017. netfilter: ipvs: Fix inappropriate output of procfs. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c5504f724c86ee925e7ffb80aa342cfd57959b13.

[80] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2018. Cntr: Lightweight OS Containers. In *2018 USENIX Annual Technical Conference (ATC '18)*. USENIX Association, Boston, MA, 199–212. https://www.usenix.org/conference/atc18/presentation/thalheim

[81] Andrei Vagin. 2017. net/unix: don't show information about sockets from other namespaces. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0f5da659d8f18.

[82] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (S&P '20)* (San Francisco, CA). 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

[83] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, and Kui Ren. 2021. Demons in the Shared Kernel: Abstract Resource Attacks Against OS-Level Virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)* (Virtual Event, Republic of Korea). Association for Computing Machinery, 764–778. https://doi.org/10.1145/3460120.3484744

[84] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. 2019. On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. 491–501. https://doi.org/10.1109/SANER.2019.8668013

[85] Liping Zhang. 2022. netfilter: conntrack: do not dump other netns's conntrack entries via proc. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e77e6ff502ea3d193872b5b9033bfd9717b36447.

[86] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. 2019. PeX: A Permission Check Analysis Framework for Linux Kernel. In *28th USENIX Security Symposium (Security '19)*. USENIX Association, Santa Clara, CA, 1205–1220. https://www.usenix.org/conference/usenixsecurity19/presentation/zhang-tong

[87] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)* (Online Event, United Kingdom). Association for Computing Machinery, New York, NY, USA, 540–555. https://doi.org/10.1145/3447786.3456258

[88] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. 2020. MPTEE: Bringing Flexible and Efficient Memory Protection to Intel SGX. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)* (Heraklion, Greece). Association for Computing Machinery, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/3342195.3387536

Received 2022-07-07; accepted 2022-09-22