2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)

TORPEDO: A Fuzzing Framework for Discovering Adversarial Container Workloads

Kenton McDonough* Virginia Tech
Xing Gao* University of Delaware
Shuai Wang HKUST
Haining Wang Virginia Tech

***Abstract*—Containers enable a computing system to host multiple isolated applications, making more cost-efficient use of the available computing resources. However, exploiting shared computing resources, adversaries can launch various real-world attacks (e.g., denial-of-service attacks) inside containers. In this paper, we present TORPEDO, a fuzzing-based approach to detecting *out-of-band* workloads: such workloads could largely interfere the performance of colocated container instances on the same host, gaining extra unfair advantages on the system resources without being charged appropriately. TORPEDO mutates inputs of OS syscalls and simultaneously monitors the resource consumption of multiple container instances. It uses resource-guided heuristics to find inputs that maximize the difference in resource consumption between container instances and resource limits. We evaluate TORPEDO on widely-used containerization platforms and demonstrate that it can verify adversarial workloads that are manually discovered by existing research. More importantly, TORPEDO identifies several zero-day vulnerabilities that are not known to the public.**

I. INTRODUCTION

Containerization platforms provide environments to isolate and provision processes running on the same host. Unlike conventional virtualization techniques maintaining an individual copy of the operating system (OS) and libraries for each instance, containers enable much more lightweight and standalone isolation toward user applications. Container instances on the same host share the same OS kernel, thus reducing computing costs by stacking kernels and driving much higher server efficiencies. To date, container techniques have been widely adopted in many scenarios including cloud computing, serverless computing, and edge computing. The value of the container market is expected to reach $8.2 billion in 2025 [3] as compared to $762 million in 2016 [64].

Containers employ system-wide isolation enforced by several kernel mechanisms provided by the host OS. In particular, various Linux kernel authorization mechanisms (e.g., namespaces and control groups) and the Linux security modules (SELinux, AppArmor, etc.) are leveraged to constitute resource isolation and provision [65]. Despite this spectacular progress, various real-world attacks have been launched to abuse the shared computing resources and affect the performance of container instances collocated with a malicious container [29], [52], [78]. For example, recently-disclosed attacks [29] have shown that by deliberately triggering system calls or locking up interrupts, Denial-of-Service (DoS) attacks can be launched toward colocated containers causing as much as 95% performance degradation.

*Co-first authors.

Given various attacks conducted on containerization platforms, previous studies reveal that many attacks are derived from the same root cause by constantly abusing the shared computing resource provision [29], [78]. The shared resources, considered as "fairly" allocated across different containers by the Linux `cgroups` mechanism, can be abused to starve the host and other colocated container instances. While many research works have manually disclosed attack strategies, a thorough and complete analysis of containerization platforms on their resource allocation enforcement is still missing in understanding today's container security landscape.

In this research, we extend the standard fuzz testing paradigm to expose provision resource drifting of container instances by stressing resources via system calls. Particularly, we capture resources being subtly manipulated by (malicious) containers; such manipulation can break the resource isolation guarantee (e.g., enforced by Linux control groups), largely consume shared system resources, and likely provoke various security attacks. Compared with existing research, our automated testing-based framework comprehensively cruises the potential attack surface of containerization platforms, and can provide counter examples (test inputs) that can actually trigger the defects. The process of discovering, debugging, and even fixing the container vulnerabilities is adequately simplified.

We have implemented the fuzzing framework into a practical and efficient tool, named TORPEDO, by addressing multiple domain-specific challenges and incorporating several optimizations in the container environments. TORPEDO is an unsupervised coverage-guided fuzzer supporting multiple containers with arbitrary resource restrictions tested on different container runtimes. It leverages resource-guided heuristics to find system call inputs that maximize the discrepancy between system resource consumption and container resource limitations. We propose to guide the fuzz testing by combining both code coverage and system resource consumption as feedback. We also propose testing oracles, with respect to the system-level resource allocation guarantees commonly assumed by the containerization platforms, to identify potential adversarial workloads.

The proposed workflow is effective and shows promising results when evaluated with Docker [11] with three different container runtime settings, including the default runc [10], the Redhat crun [24], and Google gVisor [12]. TORPEDO successfully re-confirms several vulnerabilities that are known to the community, but identifies several new attacking vectors. Also, it detects multiple new vulnerabilities concerning violations of CPU resource limitation from these popular (commercial) container components. We also trace the root cause for identified vulnerabilities in the container implementation, and investigate the potential security threats. Confirmation with container developers also receives promising feedback, with multiple findings have been promptly confirmed and fixed.

The proposed techniques and the developed platform can be adopted by virtualization technique developers and security researchers, to provide continuous updates against attackers with access to the tremendous numbers of virtualization and cloud services in the real world. To facilitate results verification and follow-up research, we will release all our erroneous detection results and a snapshot of TORPEDO.

II. BACKGROUND

To date, many container technologies are available on the market, including LXC, OpenVZ, Linux-Vserver, and Docker [11], [13]–[15]. In general, container engines such as Docker create and manage the lifecycle of containers, and container runtime is responsible for translating the visibility and resource restrictions from the user-facing API into directives for the kernel. There are several existing designs for container runtimes. Native runtimes perform the necessary setup for the container, allowing the container to share the host kernel. Examples include the default runtime packaged with Docker, runc [10], and the Red Hat crun [24], which is written in C for setting up the environment before the container process starts. Sandboxed runtimes introduce a translation layer between the container and the host kernel, and gVisor [12] is a secure runtime that reduces the attack space on the host kernel by implementing a large portion of the syscall interface with a smaller number of syscalls. At the kernel level, containers depend on multiple independent Linux kernel components (e.g., *namespace* and *cgroups*) to enforce isolation among user-space instances. Particularly, `cgroups` (i.e., control groups) are the key features for controlling and limiting the total amount of system resources for containers. We next discuss the `cgroups` mechanism in detail.

A. Linux Control Group

Modern Linux OS features `cgroups` as a highly flexible and configurable way to control the dynamic computing resource allocation, including (CPU) runtime, memory, input/output (I/O), and network bandwidth. `cgroups` quantitatively limit the amount of resources assigned to a container, thus ideally, it is designed to prevent one or a particular group of containers from draining all the available computing resources of other containers or the host machine. Typically, the `cgroups` mechanism partitions groups of processes into hierarchical groups with controlled behaviors, and relies on different resource controllers (or named as subsystems) to limit, account for, and isolate various types of system resources.

The control groups mechanism is one keystone constituting containerization platforms, enforcing both *cross-container isolation* and *container-to-host isolation* on multiple types of system resources. As mentioned above, `cgroups` specify the resource allowance for one or a set of containers. For instance, by specifying the CPU usage share of one container as 512 and another container as 1024, the latter one is provisioned to get roughly double amount of CPU time compared to the first one. Nevertheless, enforced `cgroups`, none of these two containers can starve the other one, even if they are competing the same CPU core. Similarly, `cgroups` helps to prevent containers from draining resource over the host machine. The cpu controller can provide a hard limit on the maximum amount of resource utilized by a container, by specifying a *quota* and *period*. Each container can only consume up to "quota" microseconds within each given "period" in microseconds. For a container set with 50,000 "quota" and 50,000 "period", it can consume up to the total CPU cycles of one CPU core. More importantly, `cgroups` have an inheritance mechanism, ensuing that all child processes inherit the exactly same `cgroups` attributes from their parent processes, which guarantees that all child processes will be confined under the same cgroups policies. Overall, `cgroups` provide a flexible mechanism to specify and enforce the resource quota for containers, smoothly enabling the "pay-as-you-go" scheme for real-world cloud platforms. More importantly, a correctly designed and implemented cgroup mechanism shall prevent most cross-container or container-to-host attack vectors such as Denial-of-Service (DoS) attacks in the first place.

B. Container Attacks by Abusing Resource Allocation

Despite the encouraging and flexible enforcement provided by Linux namespaces and `cgroups`, we have observed various real-world exploitations toward containerization platforms. In a multi-tenant environment where multiple containers belonging to different tenants run on the same physical machine, malicious containers might turn other co-resident containers or the host into mal-functional. For instance, a malicious container can drain most of the CPU computing resources and starve other containers or even the host OS.

Ideally, the resource consumed by a container is limited by `cgroups`. However, previous work [29], [45] demonstrates that inherited cgroups confinement via process creation cannot always guarantee consistent and fair resource accounting, and it is possible to break the resource rein of `cgroups`. Gao et al. [29] designed a set of exploiting strategies to generate out-of-band workloads on another process (in a different cgroup) on behalf of a constrained original (malicious) process. The consequence is huge: Gao et al. [29] demonstrated that, by escaping the resource limit of `cgroups`, a container can consume system resources (e.g., CPU) as much as $200\times$ of its limit, and significantly degrade the performance of other co-resident containers to only 5%.

Defer Work to the Kernel. The first type of strategy is to defer or delegate workload to the kernel, as all kernel threads are attached to the *root* `cgroup`. The amount of resources consumed by those workloads would be counted to the target kernel thread, instead of the initiating user-space process (i.e., the container). The Linux kernel by default runs multiple kernel threads, including *kworker* for handling workqueue tasks [1] and *ksoftirqd* for serving softirqs. Also, a container process can exploit kernel threads as proxies to spawn new processes (which are still attached to the root `cgroup`), and thus escape the resource control. One feasible solution is to exploit the `usermode helper` API, which provides a simple interface for creating a process in the user-space. In both cases, the corresponding consumed resources would not be limited by any `cgroups`.

Deferring Work to Other Process cgroups. The second type of strategy is to delegate workload to other userspace processes, various system daemons and services, which are all attached to other *cgroups* than a containerized process. For example, a malicious container can exploit multiple system processes (e.g., *systemd*) maintained by the Linux server for purposes like process management, system information logging, debugging, or container engine processes, which are required to run on the host to support and manage container instances. The corresponding consumed resources would not be charged to the initiating process (i.e., malicious container), and thus the `cgroups` mechanism can be escaped.

However, we consider the existing research has never fulfilled its potential by conducting a systematic and comprehensive study on resource allocation. Previous works largely rely on manual analysis and thus can only find limited exploiting methods. We thus intend to develop a system to automatically uncover those vulnerabilities in containerization platforms.

III. Problem Formulation and Approach

In this section, we formulate the research problem and discuss the opportunities to address it with fuzzing.

A. Problem Formulation

In general, containerization platforms are designed to deliver a confined provision for container instances, in terms of both static and dynamic computing resources. Container instances should not go over a pre-defined amount of static computing resources. More importantly, the provision of runtime resources should not be changed no matter how the resource is accessed by other container instances; violation of such provision may be due to bugs or inherent design limits, revealing chances of conducting exploitations.

Threat Model. We consider standard multi-tenant environments where multiple containers belonging to different tenants share the same physical machine. All containers are confined with proper resource isolation and thus can only consume limited resources (e.g., CPU cycles, memory, etc.). The attacker can control one or more containers by using the provided service normally and legitimately. The malicious container then attempts to cause system-wide impacts by consuming more resources than allocated.

Formulation. The aforementioned research problem is formulated as below. Let H represent a physical machine which hosts n container instances $\mathcal{C} = c_1, c_2, \ldots, c_n$, running with different containerization platform combinations (denoted as $\mathcal{P} = p_1, p_2, \ldots, p_s$). Once deployed, remote users can communicate with the deployed application by constantly feeding inputs and imposing one of the workloads $\mathcal{W} = w_1, w_2, \ldots, w_t$. To prevent inter-container exploitation, the container manager enforces the following holistic requirement:

$$\forall c_i \in \mathcal{C}, \forall w_j \in \mathcal{W}, \forall p_k \in \mathcal{P} : R_{i,j,k} \leq \mathit{Alloc}(c_i, w_j, p_k) \quad (1)$$

where function Alloc denotes the amount of computing resources (e.g., CPU, Memory, I/O bandwidth) provisioned for a container instance c_i, and $R_{i,j,k}$ denotes the total amount of resources consumed by that container. In general, this requirement specifies that a container should not be capable of consuming more resources than allocated by the containerization platforms.

The above formulation indicates the supposed resource consumption by each container. However, it is challenging to monitor out-of-band workloads for each container, as many processes are shared among all containers. Thus, we can make a generalization about the total resource utilization of the host (R_H) for an arbitrary set of containerized workloads. Particularly, the total resource consumed by the host should be less than the summation of allocated resources of all containers.

$$R_H \leq \sum \mathit{Alloc}(c_i, w_i, p_k) \quad (2)$$

Although satisfying the above requirement guarantees a fair resource provision, this requirement is still too strict for most real-world cases. Each OS has some amount of unavoidable overhead associated with creating and executing a containerized workload. Therefore, we use ϵ as a small drifting and refine Constraint 2 such that not only containerization platforms with strict enforcement are deemed safe, but also with small changes ϵ are safe:

$$R_H + \epsilon \leq \sum \mathit{Alloc}(c_i, w_j, p_k) \quad (3)$$

where ϵ can be configured by the users.

B. Resource-Guided Fuzz Testing

Feedback-driven fuzz testing has been widely used to automatically generate tests to detect software faults [80]. The strength of feedback-driven fuzz testing lies in its capability to benefit from the "genetic algorithm" to gradually identify and retain inputs that can maximize the fuzzing objective. We leverage the fuzzing scheme to analyze container instances running on the same host. Overall, while applications within different containers usually have different functionalities, the container instances themselves, once configured and launched, should always be confined by the specified resource provision. In that sense, if one or more containers exhibit observable violations, then it means that workloads exposed over containers provoke vulnerabilities of the tested, which should be remedied by developers.

We aim to record the violations w.r.t. the testing oracle measurements observed over container instance executions. For each iteration of testing, it mutates the test inputs to guide the fuzzer in finding inputs that *maximize* predefined feedback (in this case it is the resource allocation driftings). In general,

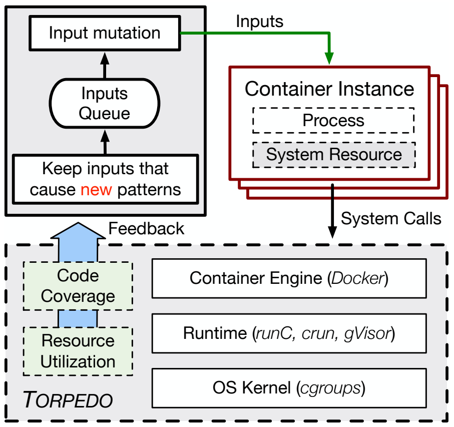


Fig. 1: Overview of the proposed approach.

feedback-driven approaches form a search campaign inspired by evolutionary biology, which aims to gradually converge test cases with high chances of success. Hence, the input creation and mutation would consider the collected feedback. It defines the requirement in this research as follows:

$$\max_{w_m} \delta = R_H - \sum Alloc(c_i, w_m, p_k) \tag{4}$$

where δ denotes the difference of allocated resources between the allocated and the real consumed. A large δ indicates a higher chance of conducting cgroup escape exploitations. As shown in Equation 4, we compute the resource allocation differences for each mutated input: inputs will be kept in a queue for further mutation and usage in case it leads to new (interesting) differences in the tested container instances.

IV. TORPEDO DESIGN

The goal of TORPEDO is to develop an unsupervised coverage-guided fuzzer supporting multiple containers to be tested on different container runtimes in parallel. Figure 1 depicts a bird's eye view. It is similar to SYZKALLER, which fuzzes pools of virtual machines, but is actually significantly different from it. Instead of spawning VM, TORPEDO creates containers with arbitrary resource restrictions and runtimes (e.g., runC, crun, gVisor) directly, thus reducing the amount of resource overhead incurred by additional isolation mechanisms in VMs.

For the general architecture, a manager binary serves as an entrypoint for the fuzzer and a central collection point for the program corpus and execution statistics. Each manager spawns a number of fuzzer processes and communicates with the fuzzers over gRPC. The fuzzer binary then runs inside a container, and is responsible for generating and manipulating programs through various lifetime stages. It repeatedly mutates programs to determine variants that generate new coverage. The executor then executes a serialized program while collecting coverage information about each call. It implements a translation layer to forward commands directly

Algorithm 1 Fuzz testing. Report all discrepant workloads across container instances $\mathcal{C}$ starting from a corpus $\mathcal{I}$. R denotes the resource (e.g., CPU cycles) for testing.

```
1:  function TORPEDO($\mathcal{I}$, $\mathcal{C}$, $R$)
2:      $\mathcal{S} \leftarrow \varnothing$                        ▷ discrepant workload set
3:      $\mathcal{O} \leftarrow$ CONFIGCONTAINER($\mathcal{C}$)
4:      for 1 ... $MAX\_ITER$ do
5:          $i \leftarrow$ POPQUEUE($\mathcal{I}$)
6:          $i^* \leftarrow$ MUTATE($i$)
7:          $\mathcal{W} \leftarrow$ GENWORKLOAD($i^*$)
8:          $\mathcal{R} \leftarrow \varnothing$          ▷ resources allocated in containers
9:          for $con_i, w_i \in (\mathcal{C}, \mathcal{W})$ do
10:             $r \leftarrow$ RUN($con_i$, $w_i$)
11:             $\mathcal{R} \leftarrow \mathcal{R} \bigcup \{r\}$
12:         if NEWPATTERN($\mathcal{R}$) then
13:             $\mathcal{I} \leftarrow \mathcal{I} \bigcup \{i^*\}$      ▷ record $i^*$ that exposes new patterns
14:         for $oracle \in \mathcal{O}$ do
15:             if VIOLATE($\mathcal{R}$, $oracle$) then
16:                 $\mathcal{S} \leftarrow \mathcal{S} \bigcup \{i^*\}$   ▷ record $i^*$ that violates an oracle
17:     return $\mathcal{S}$
```

onto the host and passes logs from the fuzzer back for analysis (i.e., between the fuzzer and executors).

TORPEDO contains an *observer*, which is a thread of execution responsible for delegating workloads to executors and examining the results of each execution. It collects a wide spectrum of system information, including various resources consumed by a container, the utilization of system/kernel processes, and the resource consumed by containerization components. For guiding adversarial program generation to identify *out-of-band* workload, TORPEDO leverage an extra library, *Oracle(s)*, that contains the necessary logic for the task with respect to a particular resource. With Oracles, TORPEDO combines both code coverage information and resource utilization to guide the fuzzing process.

Algorithm 1 specifies the testing workflow. Overall, our approach depicts a fuzz testing procedure. We start by taking a set of container instances deployed on the same host as the testing target. We also require users to provide a set of initial inputs $\mathcal{I}$ as the fuzz testing seeds (see Sec. V on the construction of $\mathcal{I}$ in our research). Containers $\mathcal{C}$ are configured w.r.t. the particular resource R and yields the corresponding testing oracle set $\mathcal{O}$ (line 3). We then iterate the fuzz testing process for *MAX_ITER* times and collect all the findings. For each iteration, n inputs are fetched from the input queue (line 5), and we mutate the fetched input set i. Then, the mutated i^* will be used to generate a set of workload $\mathcal{W}$ over each container instance (Sec. IV-A). Each container con_i will be executed with its assigned workload w_i for a reasonable amount of time (Sec. IV-B), and we collect the targeted resource computation during this phase (line 9; Sec. IV-C). In case the collected computing resource consumption reveals certain unknown patterns (e.g., a larger δ in Equation 3; see Sec. IV-D for elaboration), the mutated input i^* is considered interesting and will be kept in the input queue for further usage (line 12). More importantly, once the provoked resource usage violates any testing oracle for the checked property (line 14), we keep this input as "discrepant input". The entire set of discrepant inputs will be returned to users (e.g., container developers) for confirmation and bug fix of container runtime systems.

Our testing campaign subsumes several layers hidden within the container "runtime" (see the three layers in the "Physical Machine" box of Figure 1). In addition, it is worth noting that while we primarily detect resource violation vulnerabilities (*out-of-band* workloads), crashes hidden in the container runtime systems and libraries could also be exposed, since the proposed approach forms a typical fuzz testing toward the container infrastructure. Existing research has (manually) identified vulnerabilities of this category [18], [42], and our evaluation successfully reveals several crashes in the Docker runtime system (see Sec. VI). In the rest of this section, we elaborate on each step in detail.

A. Generating Workload

SYZKALLER does not natively support the direct ingestion of seed files for use in corpus construction. Instead, it prefers to generate programs using a nondeterministic process that may draw on a corpus of coverage information, if one is available. For more efficient testing (especially reproducing existing exploits), we envision that TORPEDO is capable of ingesting seed files directly from an operator and using these to populate an initial corpus. Given an input set i^*, TORPEDO prepares a set of workload $\mathcal{W}$ that will be used by each container instance. It then passes through a serialized execution request to a prepared container, and distributes many heterogeneous workloads in parallel. Since the majority of generated programs are short (10 ms or less) and may not finish at the same time, TORPEDO repeatedly runs those workloads and deploys a synchronization mechanism (discussed in Section IV-C) to ensure an efficient fuzzing process.

B. Interacting with the Container Runtime

In the native SYZKALLER design, workloads are executed via a virtual machine that shields the *syz-manager* binary (which serves as an entrypoint for the fuzzer and a central collection point for the program corpus and execution statistics) from kernel crashes. While TORPEDO would also benefit from such a strategy, we also note that VMs impose a nontrivial performance overhead and may obscure otherwise relevant observations. When specifically considering sandboxed and virtualized runtimes, which need to be analyzed closely for adversarial utilization on the host, adding an additional layer of VM translation will complicate detangling measurements and slow down the entire fuzzing process. Thus, we choose to execute all TORPEDO processes on the same host.

We further identify and package the smallest set of SYZKALLER components into a container to maintain the existing program execution workflow. Particularly, we package the *syz-executor* process (a C++ binary that reads in a serialized program and executes it while collecting coverage information about each call) and a simple entrypoint binary to maintain API compatibility and allow for connection debugging. Additional features of this entrypoint will be discussed in Section IV-C. These two applications, when combined, form a container image for fuzzing adversarial workloads.

C. Collecting Provisioned Computing Resources

Since the goal of TORPEDO is to identify *out-of-band* workloads that will violate existing `cgroup` limitations, it must accurately capture resource utilization measurements. To do so, the first step is to observe the state of the system while the program(s) under examination are running. Ideally, the observation window would completely overlap with the window of execution to capture an accurate measurement. This poses an issue when the programs under testing have different running times as a result of variations in the algorithmic complexity of the underlying syscalls or simply becoming blocked. Furthermore, when multiple containers are running in parallel, we note that all the programs under test will collectively contribute to the resource utilization of the host. Thus, for accurately measuring multiple fuzzing processes in parallel, we completely synchronize the program execution window and extend the execution time for each program to become comparable. We choose to have the container entrypoint binary be responsible for this synchronization. Basically we keep running the workloads in a loop until it reaches the threshold, and report the number of executions and average execution time (obtained through Unix NS timestamps). This way, TORPEDO ensures that all parallel executor containers terminate their execution at or before a specified timestamp.

Observer. To coordinate workload execution and measurement taking, we introduce the concept of the **observer**. The observer is a thread of execution responsible for delegating workloads to executors, signaling executors to start, and examining the results of each execution. These "observations" provide feedback used to guide program generation and mutation, as well as identify workloads that are likely adversarial. The observer has access to all feedback results and can use them to immediately motivate changes to each program for the following round (Section IV-D). Additionally, the observer is responsible for logging this information for later analysis (e.g., identifying adversarial workloads).

D. Constituting Fuzz Testing Feedback

TORPEDO must consider two feedback mechanisms when making decisions: *code coverage* and *resource utilization*. Code coverage constitutes a simple "binary feedback" mechanism; a given measurement either contains some new coverage or does not. A program that generates more new coverage is strictly "better" than one that does not. However, the same relationship does not necessarily hold for resource utilization: a fuzzing input that generates more CPU utilization than its predecessor may not strictly be more adversarial; it could simply spend less time blocked. For designing TORPEDO, we note that adversarial workloads typically exhibit some amount of "workload amplification", by which the total amount of resources consumed by the host is increased some factor beyond what the adversarial program is consuming itself. This indicates that observing more overall resource utilization is potentially indicative of an *out-of-band* workload, especially when resource limitations have been placed on the workloads that should restrict them. The observer thread is then responsible for collecting and analyzing this information to guide the generation of adversarial programs.

Furthermore, we split the process of guiding program generation into two separate problems. The first concerns ranking workloads with respect to "how likely" they are to become adversarial. The second concerns identifying with some certainty that a workload has become adversarial. The first functionality is necessary to motivate program mutation while evaluation is ongoing for a particular batch, and the second is necessary for the ultimate goal of identifying programs that violate one or more resource oracles. We conceive of an oracle library that contains the necessary logic for both of these tasks with respect to a particular resource $\mathcal{R}$. More formally, an oracle library must support the following operations.

1) Score a workload. A higher score indicates the workload is more indicative of adversarial behavior.
2) Flag a workload. If the flag is thrown, the oracle believes the workload violates one or more resource isolation boundaries.

The question remains of how to combine oracle and code coverage feedback in a meaningful way. Fundamentally, these two mechanisms are incompatible. Code coverage is collected per individual syscall in a program, whereas an oracle score takes into account the behavior of all programs and the host. TORPEDO solves this problem by considering both mechanisms at separate granularity levels. Particularly, code coverage is incorporated at the individual program level, and resource utilization at the "set of programs" level. In this way, the current set of all containerized workloads is considered separately from the individual workloads that comprise the set.

V. IMPLEMENTATION

A. *Instrumenting* SYZKALLER

The OS kernel fuzzing framework SYZKALLER [32], [56] takes a set of system call traces (each set is called a "corpus") as its seed inputs for fuzzing. Given a corpus of system call traces, SYZKALLER perturbs input values of system calls and also shuffles system calls on the trace to interact with the OS kernel. SYZKALLER can also generate new traces during the fuzzing campaign. It manifests a standard feedback driven grey-box fuzzing setting guided by kernel code coverage. A trace is kept for further mutations if the executing system calls on the trace induces new coverage of the OS kernel; otherwise, it is discarded. In TORPEDO, we instrument SYZKALLER and take the resource consumption difference among container instances as another feedback to guide fuzzing. The whole implementation contains 1,500+ Go codes as well as non-trivial C/C++ modifications.

Replacing Virtual Machines with Containers. We begin by implementing a VM translation layer that creates processes on the host and passes logs from the fuzzer back for analysis. Execution requests are passed to containers via IPC pipes and results are returned using the same mechanism. We also introduce a small library to support creating containers with arbitrary resource restrictions and runtimes. Rather than directly interacting with the Docker daemon over HTTP, we implement a wrapper around the Docker command line interface. This ensures that TORPEDO is capable of capturing potential vulnerabilities created by the interaction between the CLI and the Docker Daemon, as well as compatible with equivalent container engines like "podman", which use the same CLI commands. Each container is restricted via cgroup constraints to a single, unique physical core, which makes it easier to identify when a containerized workload has "escaped" to another core (i.e., breaking the `cpuset` cgroup).

Algorithm 2 Observe Execution. Each round lasts for $\mathcal{T}$ seconds. $\mathcal{R}$ represents some computing resource the observer should monitor.

```
1: function OBSERVER($\mathcal{T}$, $\mathcal{R}$)
2:     $RoundNum \leftarrow 0$
3:     $Workloads \leftarrow \varnothing$
4:     $RoundScore \leftarrow 0$
5:     INITIALIZEEXECUTORS($\mathcal{E}$)
6:     for $\infty$ do
7:         $\mathcal{W} \leftarrow$ GETPROGRAMS($\mathcal{W}$, $RoundScore$, $\mathcal{R}$)
8:         $StopTime \leftarrow CurrentTime + \mathcal{T}$
9:         for $\mathcal{E} \in Executor$ do
10:            $\mathcal{E}.stop \leftarrow StopTime$
11:            $\mathcal{E}.program \leftarrow w\ (w \in \mathcal{W})$
12:            SIGNAL($\mathcal{E}$)
13:        WAITFORALLEXECUTORS($\mathcal{E}$)    ▷ Wait for all executors to signal they are ready
14:        SIGNALALLEXECUTORS($\mathcal{E}$)
15:        $RoundScore \leftarrow$ TAKEMEASUREMENT($\mathcal{T}$, $\mathcal{R}$) ▷ returns after $\mathcal{T}$ seconds
16:        LOGROUNDRESULTS($RoundScore$)
17:        $RoundNum{+}{+}$
18:
19: function EXECUTOR($\mathcal{O}$)    ▷ Each executor maintains a reference to the observer
20:     $program \leftarrow \varnothing$
21:     $stop \leftarrow \varnothing$
22:     for $\infty$ do
23:         WAITFORSIGNAL()
24:         PREPARETOEXECUTE($program$)    ▷ Create a container and serialize execute request
25:         SIGNALOBSERVER($\mathcal{O}$)
26:         WAITFORSIGNAL()
27:         EXECUTE($program$, $stop$)
```

Implementing the Observer. In the SYZKALLER native design, one thread is created for each executor and all procs execute independently. We modify each thread and coordinate the observer with any number of executors using algorithm 2. Basically, this algorithm uses a two-stage latching procedure to distribute programs and prime each executor (each executor is distributed with one program, as Line 11), then starts the execution window to align with a pre-defined number of resource measurements.

The observer divides execution periods into rounds of time T duration each. With preliminary exploration, we observe that a short interval (T) is more susceptible to being disrupted by temporary "noise spikes" from the host (e.g., cron jobs, sudden arrival of network packets, system logging events, etc), whereas longer intervals produce more useful measurements but significantly reduce program throughput. We choose values in the range of a few full seconds, often between three and five, to achieve a fair balance of throughput and precision.

Implementing Oracles. In anticipation of fuzzing for adversarial CPU utilization, we create an oracle framework suitable for collecting both per-process and per-core utilization measurements (Line 15 of algorithm 2, *TakeMeasurements*). The former can be easily collected from the PROC file system, specifically through '/proc/stat'. This pseudofile exposes information about how much time each CPU core has spent in various categories, including userspace, kernel space, and idling.

Collecting per process CPU utilization is more difficult, but can provide equally useful insights. Tracking the usage of individual processes is particularly helpful in understanding where out-of-band workloads are being created and tracking their efficacy. To implement this, we fork an existing Golang library [7] with a wrapper for the `top(1)` command. We filter this output by selecting common categories of interest, such as 'docker', 'kworker' threads, 'kauditd', 'systemd-journal', and miscellaneous kernel threads (most of them are reported in [29]).

The implementation of top on Linux has a number of hidden idiosyncrasies that make it difficult for our purpose. First, even when invoked with a custom duration between updates, top has an unavoidable "warm up time" to generate its first frame that produces inaccurate results. We modify the Linux wrapper for top to discard these warm-up measurements. Secondly, top is incapable of reporting CPU utilization by processes that begin or end during the time between frames. For our purposes, this only makes it suitable for measuring CPU utilization from daemons or otherwise long-lived processes. If a program were to trigger the creation of many short lived kernel threads, TORPEDO would still observe it from the broader per-core CPU usage measurement. The combination of these two metrics gives an excellent "snapshot" of CPU allocation during a time period and can easily be analyzed to determine adversarial workloads.

B. Leveraging the Oracle Library

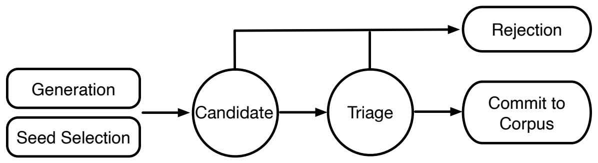
As in Section IV-D, we implement each oracle to support two objectives. The first concerns scoring workload resource utilization to serve as a feedback mechanism, and the second concerns identifying adversarial workloads based on a set of heuristics.

Scoring Workloads. As in SYZKALLER, candidate programs are evaluated for new code coverage patterns and only accepted for triage if they are judged to be interesting. Each batch of programs is subjected to many repeated mutations in an attempt to motivate the generation of adversarial programs. We conceive of two states that a set of programs may be in at any time; "mutation", where each program in the set is perturbed in an attempt to generate more adversarial resource utilization, and "confirm", where programs are rerun to confirm some interesting observation exists and was not a result of system noise. The Oracle score is used to determine when a mutation has achieved some meaningful change and should be confirmed as a new baseline for the batch (Algorithm 2, Line 15, *RoundScore*, used to *GetPrograms* on line 7). After some amount of time without a meaningful improvement, the Oracle determines the batch has been exhausted and calls for new programs.

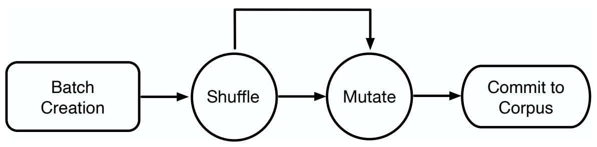
Combining Coverage and Utilization Feedback. As the primary assumption behind most fuzzing tools, high code coverage generally means that it is more likely for a test to uncover a bug. Nevertheless, this might not be sufficient for our focus, aiming at finding bugs enabling adversarial workloads. Thus, TORPEDO needs to combine both code coverage information and system utilization feedback to guide the fuzzing process. This is not straightforward, because code overage comes from an individual program but system utilization comes from all programs. To this end, our design splits the SYZKALLER program state machine into two separate state machines: one for each program and the other for the whole batch of programs. Figure 2 depicts the result of dividing relevant states between the level of an individual program and a batch of programs. The program state machine is focused on coverage collection: it discards programs that are not interesting and ensures to keep getting new traces to test. The batch state machine is focused on the system utilization: it decides how to mutate programs. Thus, programs that do not generate new coverage are typically rejected before they spend too much time being mutated. Also, only the set of mutated workloads that generate the most adversarial resource usage are recorded into the corpus.

To reduce the impact of system noise (e.g., generated by mutation operations), we implement the "shuffle" state, where individual programs are shuffled between cores but the order of syscalls in each trace remains unchanged. This helps to reduce false positives from the scenarios where system noise is concentrated on a subset of cores and is unrelated to the program under test.

Flagging Workloads. The Observer could easily apply an Oracle's flagging heuristic to each observation as it becomes available, although as true violations are likely to be rare, this would reduce overall program throughput. Instead, TORPEDO uses this Oracle functionality to parse through log files from each round and isolate small numbers of adversarial programs



(a) Program State Machine.



(b) Batch State Machine.

Fig. 2: TORPEDO State Machines

asynchronously from program execution. If the adversarial program is indeed correlated with a higher score from the Oracle's *scoring* functionality, then we expect the adversarial program to be retained for the remainder of the batch, which serves to confirm the program is the cause of whatever adversarial behavior has been flagged. Once identified, TORPEDO leverages a tool-assisted minimization workflow to automatically isolate the adversarial programs (i.e., a sequence of system calls) for further analysis. Basically, we systematically remove calls from the program until we obtain the smallest set of calls that result in the originally observed oracle violations. After that, we further manually confirm and isolate the vulnerabilities through kernel trace debugging.

VI. EVALUATION

A. Research Questions

TORPEDO is designed to discover vulnerabilities existed in containerization components, which can be exploited to generate out-of-band workloads and escape the resource limit of cgroups. While many aspects of the tool are novel, much of the design is a natural extension of the existing SYZKALLER framework. Also, we source much of TORPEDO's initial testing corpus directly from a selection of seeds from Moonshine [56], which is another SYZKALLER extension project concerned with improving the quality of seeds distilled from the framework seeds. Particularly, we attempt to explore that (1) Can TORPEDO discover new vulnerabilities and how efficient is that? (2) Are there discrepancies among different implementations of container components (e.g., runtime)? (3) How is TORPEDO's code coverage mechanism?

B. Environment Setups

Multiple popular container implementations are commonly used in the real world. This paper aims at presenting an in-depth understanding of today's container security landscape, where we will leverage TORPEDO to test three popular container runtime implementations: runc, crun, and gVisor.

TORPEDO is designed as a blackbox testing framework that does not rely on any implementation details of the underlying container implementation. In principle, the proposed technique can be smoothly migrated to test different container infrastructure implementations. We consider this as a big advantage, compared to existing container security analysis techniques where heavyweight program analysis methods are conducted [57]. The following paragraphs discuss each of the container runtimes we target in detail.

runc. This runtime is used in a typical container execution environment (usually as the default container runtime), where the native Linux together with `runc` [10] are executed directly on top of the hardware. `runc` denotes a low-level container runtime library mainly supporting "high-level" container engine (e.g. Docker) to spawn and run containers. For instance, the Docker engine leverages `runc` to handle tasks such as running a container, attaching a process to an existing container, and so on.

crun. Much like `runc` [24], `crun` is a bare-metal runtime that interfaces directly with Linux to create a containerized process. Unlike runc, which is written in Golang, crun is written entirely in C. The project authors advertise that crun functions identically to runc, but due to its implementation, is faster and more memory efficient. Also, the `crun` is completely compatible with Docker.

gVisor. This runtime is indicative of another popular container execution environment, where the gVisor `(runsc)` process serves as a secure sandbox for the untrusted containerized code. gVisor functions as a userspace kernel (comparable to LibOS [9]) with a subtle difference. gVisor essentially provides an extra layer between the container and host OS kernel, intercepting system calls made by the containerized applications. To date, gVisor has successfully supported 211 out of 319 x86-64 Linux system calls, by using only 64 system calls on the host system [12].

For the duration of our experiments, we fix TORPEDO to use the Docker ecosystem with a selected container runtime. By rotating adversarial programs between different runtimes, TORPEDO empowers testers to quickly identify discrepancies between each implementation, as well as expose underlying OS bugs or higher-level bugs in Docker.

C. Evaluation Procedure

For each fuzzer execution, we choose a small number of Moonshine seeds and use TORPEDO's seed ingestion workflow to enqueue these as candidate programs. We allow the fuzzer to run unattended and review the execution logs after all seeds have been exhausted. The corpus of coverage information is purged between each fuzzer invocation, which serves to prevent adversarial system call traces from being continually injected into future programs and preventing new, interesting findings from being revealed.

Each execution encompasses some number of rounds, each of which produces a detailed log file of resource utilization during the period. These log files are batched and passed over by an automated script that examines each round for resource pattern violations as defined by one or more Oracles. This work focuses primarily on the results from a CPU Oracle, which uses the heuristics given in Table I. Specific constants for each

TABLE I: TORPEDO CPU Oracle Heuristics

Heuristic	Notes
Fuzzing core CPU utilization	Expect above some threshold
Idle core CPU utilization	Expect below some threshold
Total CPU utilization	Expect below some threshold
System process CPU utilization	Expect below some threshold

heuristic vary according to test parameters, and specifically the selected container runtime and amount of parallelism.

Due to the relationship between patterns of adversarial CPU utilization and an increased score from the CPU Oracle, we assume that the adversarial properties of a program will be preserved by TORPEDO during operation and will exhibit the same patterns over many subsequent rounds. Therefore, for a given batch of programs, any commonalities between programs flagged by the Oracle for similar resource violations can be extracted and minimized by a human operator with little difficulty. We consider any set of system call traces that creates an adversarial workload when isolated and run independently of the TORPEDO framework to be a discovered vulnerability for the purposes of evaluation.

All tests were executed on a machine equipped with an AMD Ryzen 3600X with 12 cores and 16 gigabytes of RAM running Linux kernel version 5.8 (Ubuntu 20). We run 3 containers in parallel: each is pinned to one core and restricted with 100% CPU utilization of one core, using the `cpusets` and `cpu` controllers following [29]. For the fuzzing process, each round lasts 5 seconds, and multiple batches (each typically contains between 30 and 50 rounds) are conducted. The specific number depends on whether the seed is interesting or not.

D. Summary of Identified Vulnerabilities

Table II presents a summary of our fuzzing results. We also report a computed "amplification factor" as [29], which defines the difference between the CPU utilization measured on the container cgroup (via `docker stat`) and the actual system utilization. Over the course of our testing, TORPEDO identifies three new vulnerabilities concerning CPU utilization with different attack vectors for baremetal runtimes (e.g., runc and crun). The first involves a technique that exploits the kernel module loading system to create processes outside the cgroup of the caller. The second allows a privileged container to directly place work on a kernel workqueue. The third involves manipulating a container into a state that will cause the Docker daemon to expend significant resources when the container is reclaimed by the system. Also, sandbox runtime incurs less problems compared with baremetal runtimes. To the best of our knowledge, these vulnerabilities are not currently documented in the literature.

TORPEDO also identifies several new attacking vectors belonging to known vulnerability categories, such as several new ways to trigger coredump, which can be exploited to amplify more than 200x workloads [29]. Also, it identifies bugs that can cause container crashes on gVisor, and another type of problems causing unwanted seccomp logs on crun. Both were confirmed and fixed by the developers [5], [8].

Based on the breadth of these findings, we conclude that TORPEDO is capable of identifying both known and new vulnerabilities. Note that the SYZKALLER authors have already spent some time fuzzing gVisor (not through a container interface). TORPEDO can still uncover new bugs. A detailed discussion of these findings is presented in Section VII.

E. Comparison of Code Coverage and Program Throughput

While TORPEDO and SYZKALLER are designed for different purposes with a different feedback collecting mechanism, the code coverage mechanism is similar. In general, SYZKALLER is capable of collecting much more coverage than TORPEDO over a given time delta, as all components execute asynchronously. Furthermore, SYZKALLER can more quickly screen out candidate programs that do not produce new coverage. TORPEDO is fundamentally slowed by the nature of its observation mechanisms, which requires both expensive synchronization and repetition. We run TORPEDO using different round times (e.g., 3 and 5 seconds) for 16 hours. We collect two metrics from each experiment: the number of executed programs and the amount of coverage generated. For ease of comparison, we choose to fuzz gVisor, which does not allow granular coverage collection. The "Coverage" refers to the number of unique combinations of syscall number and error code.

We observe that TORPEDO is magnitudes slower at collecting coverage than stock SYZKALLER. For example, TORPEDO with 3s rounds takes about 10 hours to reach a similar coverage as SYZKALLER for fuzzing one hour on our testbed. This is directly related to the difference in the number of programs executed per time delta. SYZKALLER executes 20,000 programs in less than an hour, whereas TORPEDO takes nearly six hours. To some extent, this gap is also exacerbated by differences in the coverage collection algorithm. SYZKALLER requires at least one program execution per syscall in a given candidate program to confirm coverage, whereas TORPEDO examines coverage for each syscall in a given program at once. In this respect, TORPEDO is marginally more efficient than SYZKALLER. However, after running TORPEDO for enough time, it can achieve similar coverage. While it is a magnitude slower than SYZKALLER at accumulating coverage, this downtime is necessary to collect additional feedback that allows TORPEDO to identify more bugs than kernel crashes.

VII. DISCUSSION

A. Confirmation of Existing Resource Vulnerabilities

Over the course of testing, TORPEDO independently reconfirms all vulnerabilities already known to the community [29]. For example, the tool can identify calls flushing data from containerized programs to a TTY on the host, which causes additional utilization on dockerd and containerd. Also, the kernel core-dump mechanism creates out-of-band workload in userspace for each core dump produced by a containerized process. Basically, the core-dump code in the kernel invokes a user-space application via the usermode helper API. The resource consumed by the user-space application will be charged to the kernel, instead of the initiating container. This

TABLE II: Summary of Uncovered Vulnerabilities

Concerned syscall	Runtimes	Attack vector	Amplification factor	Notes
socket	runc, crun	modprobe via usermodehelper	120x	ERRNO 93, 94, 97
unshare	runc, crun	kworker queue	2x	CLONE_NETNS, requires NET_ADMIN
mkdir	runc, crun	docker daemon	N/A	reclaiming container stresses dockerd
mount and others	crun	seccomp unwanted logs	2.3x	any usage
rt_sigreturn	runc, crun	coredump via SIGSEGV	200x	any usage
rseq	runc, crun	coredump via SIGSEGV	200x	invalid arguments
fallocate, ftruncate	runc, crun	coredump via SIGXFSZ	200x	argument exceeds max file size
open	gVisor	invalid argument	N/A	container crash

can amplify the workload more than 200x. Although the attacking vector is known, TORPEDO is still able to disclose several system calls with particular usages (shown in Table II) tripping the core-dump mechanisms.

B. *Case Study: sockets*

TORPEDO observes consistent workload placed on non-executor cores when programs contained certain socket related calls, which means that both cpu and cpuset cgroups are escaped. Through tool-assisted minimization, we discover that the socket syscall produces adversarial behavior. An investigation into the implementation of socket(2) by the kernel revealed a new attack vector exploiting the kernel module hot-loading mechanism. In particular, when a process creates a socket, it can specify a wealth of options including socket type, protocol, and address family. Some of these options, however, are invalid in certain combinations, or when the kernel has not been compiled to support them. Frequently, specific socket implementations are compiled as modules. When the kernel receives a request for a socket option it understands but has no implementation for, it will attempt to load a module from disk by means of the modprobe(8) tool. On success, this module is loaded into kernel memory for the remainder of execution. On failure, an error is returned to the caller. The modprobe(8) tool is invoked in userspace via the problematic usermode helper API, which changes the invoked program to the kernel's cgroup as part of execution [29].

Repeated requests from userspace for a socket that triggers an unsuccessful module hot-load create a significant out-of-band workload in userspace. With a reasonably optimized single threaded implementation, we find that an adversarial container allocated just *0.2%* of the CPU on our testbed can cause an overall system utilization of 10%. This corresponds to an amplification factor of close to 120x. Also, TORPEDO discovered several variations of arguments to socket(2) that would trigger the vulnerability, corresponding to errno 93, 94 and 97 respectively.

C. *Case Study: unshare*

TORPEDO observes a significant spike in kthread utilization when repeated invoking unshare(CLONE_NETNS) in privileged containers (i.e., executing unshare(2) on the NET namespace), which directly correlates to the creation of a work item on a kernel work queue. The adversarial effects of unshare(2) have already been noted with respect to increasing container startup time [55], as the creation of a new net namespace requires holding a global lock [4]. We also demonstrate that it can be exploited to generate adversarial *out-of-band* workloads to consume extra CPU utilization. A naive program on our testbed can cause an amplification factor of more than 2x. Also, a large amount of system memory is consumed by the unsharing processes during the test.

TABLE III: dockerd utilization per number of directories

Number of dirs	dockerd utilization (percent of entire CPU)
20k	15%
40k	24%
100k	25%

D. *Case Study: file systems*

TORPEDO consistently observes increased utilization from the docker daemon when test programs contained calls to mkdir(2). Particularly, this behavior is correlated to the period where the container is torn down. Subsequent experiments with destroying containers that contain many directories yielded the observations in Table III. On our testbed, dockerd committed three threads to tear down a container with many directories, leading to an *out-of-band* workload of 25% with 100k dirs. This workload persisted linearly with an increase in the number of directories in the container: 40k directories took approximately two seconds, but 100k directories took far longer. We note that the notion of an "amplification factor" does not make sense for this vulnerability, as the process that "caused" the utilization is no longer running when the workload manifests.

Troublingly, this workload is still out of band, as restricting the CPU, I/O or PID limitations on the container does not limit this workload. We conceive of a potential attack whereby a malicious user creates many such containers that contain millions of directories across a series of hosts. These "landmine" containers are relatively harmless until the user allows the container engine to reclaim them, at which time the docker daemon will degrade the performance of all workloads running on the host until reclamation is complete. To the best of our knowledge, this is the first potential attack that exploits the container teardown process to escape cgroup limitations.

E. *Bugs*

Through the course of testing, TORPEDO discovers several bugs in the crun and gVisor runtimes respectively. While fuzzing crun, TORPEDO identified periodic utilization spikes on the Linux audit subsystem (kauditd and journald) from seemingly unrelated program traces. The examination determines these spikes occurred from overzealous logging stemming from crun's interpretation of the Docker default seccomp profile [8]. It can cause an amplification factor about 2.3x. This bug was promptly addressed by the crun team.

While fuzzing gVisor, TORPEDO repeatedly detects a container crash across many different seeds. During minimization, a commonality is unearthed concerning the open(2) syscall and a specific argument bitfield combination. This would cause gVisor to translate a syscall not allowed by its own seccomp profile, and the container would be terminated [5]. This bug was recognized by the gVisor team and promptly fixed.

F. Future Work

While this work mainly concentrates on finding vulnerabilities related to the CPU resource, TORPEDO can be extended to uncover potential issues in other `cgroups` subsystems (e.g., memory). We will explore this direction in our future work.

VIII. RELATED WORK

In this section, we review existing research efforts that inspire our work. We mainly focus on the following areas:

Container Security. Containers typically have better performance than traditional VMs [27], [54] and thus can support real-time applications [62]. Meanwhile, container security has also received much attention. Several previous research efforts, including Gupta [34], Bui [18], and Grattafiori et al. [33], have presented a brief analysis of *Docker* security in terms of the isolation and corresponding kernel security mechanisms. Particularly, it has been demonstrated that some of the existing exploits can successfully launch attacks from inside the container [50]. Gao et al. [28], [30] also investigated the information leakage problem and its security implications in Linux containers caused by problems in *namespaces*. Multiple works have also been proposed to secure containers. Lei et al. proposed to reduce the number of available system calls to applications [48]. Sun et al. [65] proposed new security namespaces enabling autonomous security control for containers, and Arnautov et al. [16] secured Linux containers using Intel SGX.

In terms of the security problems in resource control, Gao et al. [29], combined with other previous research [45], have disclosed that particular workloads can generate extra *out-of-band* workloads than the limit of `cgroups`, which can further slow down the container [46], [88]. Yang et al. further demonstrated that the shared kernel variables and data structure can cause DoS attacks against other containers [78]. Liu et al. also discovered significant performance variations in container-based multi-tenant environments for CPU utilization. Our work further attempts to systematically explore the security problems related to container resource control.

Security Testing of System Software. Typical system software are highly complicated software with millions of lines of code, complex program structures, deep call hierarchies, and also stateful execution models. To date, fuzz testing has been commonly used to pinpoint vulnerabilities residing within the system software [74] due to its automated nature. In recent years, there has been growing research interest on fuzzing in both industry and academia [2], [6], [69]. Particularly, *Grey-Box fuzzers* [19], [22], [58] use lightweight instrumentation to track program coverage for each input without requiring extensive knowledge of the target application. Kernel fuzzing [23], [35], [41], [47], [60] has been an important topic as vulnerabilities in kernel code cause serious security breaches, from information leakage to privilege escalation. One key strategy in kernel fuzzing is to utilize types and dependencies of system calls (syscalls). Google has developed SYZKALLER [32] as an unsupervised coverage-guided kernel fuzzer specified for fuzzing operating system kernels through the system call interface. Recent research works [35], [56], [68] perform advanced analysis (e.g., reinforcement learning) to synthesize system call traces of high coverage. In addition, recent research works also conduct security fuzz testing of OS drivers and file systems [43], [61], [74]. Under most circumstances, the majority of the work surrounding fuzzing is related to isolating crashes. This perspective, however, fails to take into account the wide range of behaviors that do not result in errors or crashes exhibited by a program, which also motivates our work. TORPEDO instead focuses on finding out-of-band vulnerabilities in containers, and takes resource utilization as fuzzing feedback for the first time.

Cloud Security and Side/Covert Channel Attacks. Resource sharing facilitates cloud platforms by improving hardware utilization and reduce cost. Nevertheless, various real-world attacks have been launched to abuse the shared computing resource and affect the performance of cloud service users co-located with a malicious user [21], [36]. In typical clouds, attackers can place malicious VMs co-resident with targets on the same server [59] and then launch various attacks (e.g., side-channel [26], [44], [51], [79] and covert-channel attacks [25], [63]). Zhang et al. demonstrated that it is feasible to launch real side-channel attacks on the cloud [73], [84], [85]. Methods like last level cache [75], memory bus [70], memory deduplication [71], core temperature [17], [53] are effective for covert-channel construction. While multiple defense mechanisms have also been proposed [20], [72], [81], [83], [86], [87], it is still possible to achieve co-residence in existing mainstream cloud services [67], [76]. With the shared underlying computing resources, DoS attacks are thus possible in clouds, including resource-freeing attacks [66], memory DoS attacks [82], I/O exhausting attacks [37]. Moreover, multiple attacks [31], [38]–[40], [49], [77] attempt to exhaust the shared infrastructures (e.g., power facility) to launch DoS attacks. As the insufficiencies in cgroups could also be exploited to launch multiple attacks (e.g., covert channel / DoS) [29], our work can help mitigate potential threats in clouds.

IX. CONCLUSION

We have presented TORPEDO, a fuzz testing framework that aims to detect *out-of-band* workloads in containerization platforms that can abuse the system resource allocation and gain extra unfair advantages. TORPEDO leverages resource-guided heuristics to find system call inputs that maximize the unfairness in system resource consumption across container instances. Our evaluation confirms vulnerabilities in popular containerization platforms which were found with manual efforts. We also identify several vulnerabilities that are unknown to the public.

Acknowledgements

We would like to thank the anonymous reviewers and the shepherd, Laurent Réveillère, for their insightful and constructive comments, which helped us to improve the quality of this paper. This work was supported in part by the U.S. National Science Foundation (NSF) grant CNS-2054657 and Army Research Office (ARO) grant W911NF-19-1-0049.

References

[1] 451 Research: Application containers will be a $2.7bn market by 2020. https://451research.com/images/Marketing/press_releases/Application-container-market-will-reach-2-7bn-in-2020_final_graphic.pdf.

[2] american fuzzy lop. https://github.com/google/AFL.

[3] Application Container Market Worth $8.2 Billion by 2025. https://www.prnewswire.com/news-releases/application-container-market.html.

[4] cleanup_net is slow. https://lkml.org/lkml/2017/4/21/533.

[5] Hostfilemapper.regeneratemappings trips over seccomp rules. https://github.com/google/gvisor/issues/6116.

[6] libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[7] Linux Inspect. https://github.com/gyuho/linux-inspect.

[8] unwanted seccomp log actions stressing kauditd/journald. https://github.com/containers/crun/issues/683.

[9] LibOS. https://lwn.net/Articles/637658/, 2014.

[10] runc. https://github.com/opencontainers/runc, 2019.

[11] Docker. https://www.docker.com/, 2020.

[12] gVisor. https://gvisor.dev/, 2020.

[13] Linux Vserver. http://www.linux-vserver.org/Welcome_to_Linux-VServer.org, 2020.

[14] LXC. https://linuxcontainers.org/lxc/introduction/, 2020.

[15] OpenVZ. https://openvz.org/, 2020.

[16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *USENIX OSDI*, 2016.

[17] Davide B Bartolini, Philipp Miedl, and Lothar Thiele. On the Capacity of Thermal Covert Channels in Multicores. In *ACM EuroSys*, 2016.

[18] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[19] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE S&P*, 2018.

[20] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B Lee, Haibo Chen, and XiaoFeng Wang. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *ACM AsiaCCS*, 2018.

[21] R. C. Chiang, S. Rajasekaran, N. Zhang, and H. H. Huang. Swiper: Exploiting Virtual Machine Vulnerability in Third-Party Clouds with Competition for I/O Resources. *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[22] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-Box Concolic Testing on Binary Code. In *IEEE/ACM ICSE*, 2019.

[23] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface Aware Fuzzing for Kernel Drivers. In *ACM CCS*, 2017.

[24] Giuseppe Scrivano Dan Walsh, Valentin Rothberg. An introduction to crun, a fast and low-memory footprint container runtime. https://www.redhat.com/sysadmin/introduction-crun, 2020.

[25] Dmitry Evtyushkin and Dmitry Ponomarev. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *ACM CCS*, 2016.

[26] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. Branchscope: A New Side-Channel Attack on Directional Branch Predictor. In *ACM ASPLOS*, 2018.

[27] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *IEEE ISPASS*, 2015.

[28] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *IEEE/IFIP DSN*, 2017.

[29] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In *ACM CCS*, 2019.

[30] Xing Gao, Benjamin Steenkamer, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. A Study on the Security Implications of Information Leakages in Container Clouds. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[31] Xing Gao, Zhang Xu, Haining Wang, Li Li, and Xiaorui Wang. Reduced Cooling Redundancy: A New Security Vulnerability in a Hot Data Center. 2018.

[32] Google. Syzkaller. https://github.com/google/syzkaller, 2018.

[33] Aaron Grattafiori. NCC Group Whitepaper: Understanding and Hardening Linux Containers, 2016.

[34] Udit Gupta. Comparison between Security Majors in Virtual Machine and Linux Containers. *arXiv preprint arXiv:1507.07816*, 2015.

[35] HyungSeok Han and Sang Kil Cha. IMF: Inferred Model-based Fuzzer. In *ACM CCS*, 2017.

[36] Y. Han, J. Chan, T. Alpcan, and C. Leckie. Using Virtual Machine Allocation Policies to Defend against Co-Resident Attacks in Cloud Computing. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[37] Qun Huang and Patrick PC Lee. An Experimental Study of Cascading Performance Interference in a Virtualized Environment. *ACM SIGMETRICS*, 2013.

[38] Mohammad A Islam and Shaolei Ren. Ohm's Law in Data Centers: A Voltage Side Channel for Timing Power Attacks. In *ACM CCS*, 2018.

[39] Mohammad A Islam, Shaolei Ren, and Adam Wierman. Exploiting a Thermal Side Channel for Power Attacks in Multi-Tenant Data Centers. In *ACM CCS*, 2017.

[40] Mohammad A Islam, Luting Yang, Kiran Ranganath, and Shaolei Ren. Why Some Like It Loud: Timing Power Attacks in Multi-Tenant Data Centers Using an Acoustic Side Channel. *ACM SIGMETRICS*, 2018.

[41] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *IEEE S&P*, 2019.

[42] Zhiqiang Jian and Long Chen. A Defense Method against Docker Escape Attack. In *International Conference on Cryptography, Security and Privacy*, 2017.

[43] Zu-Ming Jiang, Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Fuzzing Error Handling Code in Device Drivers Based on Software Fault Injection. In *ISSRE*, 2019.

[44] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A High-Resolution Side-Channel Attack on Last-Level Cache. In *IEEE DAC*, 2016.

[45] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating Network-Based CPU in Container Environments. In *Usenix NSDI*, 2018.

[46] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-Based Virtual RDMA Networking for Containerized Clouds. In *USENIX NSDI*, 2019.

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

[48] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. Speaker: Split-Phase Execution of Application Containers. In *Springer DIMVA*, 2017.

[49] Chao Li, Zhenhua Wang, Xiaofeng Hou, Haopeng Chen, Xiaoyao Liang, and Minyi Guo. Power Attack Defense: Securing Battery-Backed Data Centers. In *IEEE ISCA*, 2016.

[50] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *ACM ACSAC*, 2018.

[51] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P*, 2015.

[52] Yang Luo, Wu Luo, Xiaoning Sun, Qingni Shen, Anbang Ruan, and Zhonghai Wu. Whispers Between the Containers: High-Capacity Covert Channel Attacks in Docker. In *2016 IEEE Trustcom/BigDataSE/ISPA*, 2016.

[53] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal Covert Channels on Multi-core Platforms. In *USENIX Security*, 2015.

[54] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. Lightweight Virtualization: A Performance Comparison. In *IEEE IC2E*, 2015.

[55] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *USENIX ATC*, 2018.

[56] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *UNISEX Security*, 2018.

[57] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. New Directions for Container Debloating. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, 2017.

[58] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-Aware Evolutionary Fuzzing. In *NDSS*, 2017.

[59] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *ACM CCS*, 2009.

[60] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security*, 2017.

[61] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *NDSS*, 2019.

[62] Cristian Constantin Spoiala, Alin Calinciuc, Corneliu Octavian Turcu, and Constantin Filote. Performance Comparison of a WebRTC Server on Docker versus Virtual Machine. In *IEEE International Conference on Development and Application Systems*, 2016.

[63] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural Minefields: 4k-Aliasing Covert Channel and Multi-Tenant Detection in IaaS Clouds. In *NDSS*, 2018.

[64] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access*.

[65] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. Security Namespace: Making Linux Security Frameworks Available to Containers. In *USENIX Security*, 2018.

[66] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M Swift. Resource-Freeing Attacks: Improve Your Cloud Performance (At Your Neighbor's Expense). In *ACM CCS*, 2012.

[67] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *USENIX Security*, 2015.

[68] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *USENIX Security*, 2021.

[69] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-Aware Greybox Fuzzing. In *IEEE/ACM ICSE*, 2019.

[77] Zhang Xu, Haining Wang, Zichen Xu, and Xiaorui Wang. Power Attack: An Increasing Threat to Data Centers. In *NDSS*, 2014.

[70] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security*, 2012.

[71] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security Implications of Memory Deduplication in a Virtualized Environment. In *IEEE/IFIP DSN*, 2013.

[72] Qiuyu Xiao, Michael Reiter, and Yinqian Zhang. Mitigating Storage Side Channels Using Statistical Privacy Mechanisms. In *The ACM Conference on Computer and Communications Security (CCS)*, 2015.

[73] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security*, 2016.

[74] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *IEEE S&P*, 2019.

[75] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *ACM CCSW*, 2011.

[76] Zhang Xu, Haining Wang, and Zhenyu Wu. A Measurement Study on Co-residence Threat inside the Cloud. In *USENIX Security*, 2015.

[78] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, et al. Demons in the Shared Kernel: Abstract Resource Attacks Against OS-level Virtualization. In *ACM CCS*, 2021.

[79] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*, 2014.

[80] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2017.

[81] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A Real-Time Side-Channel Attack Detection System in Clouds. In *Springer RAID*, 2016.

[82] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. DoS Attacks on Your Memory in Cloud. In *ACM AsiaCCS*, 2017.

[83] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *IEEE S&P*, 2011.

[84] Yinqian Zhang, Ari Juels, Michael Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM CCS*, 2012.

[85] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *ACM CCS*, 2014.

[86] Yinqian Zhang and Michael Reiter. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *ACM CCS*, 2013.

[87] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *ACM CCS*, 2016.

[88] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim:OS Kernel Support for a Low-Overhead Container Overlay Network. In *USENIX NSDI*, 2019.