TORPEDO: A Fuzzing Framework for Discovering Adversarial Container Workloads

Kenton R. McDonough

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Haining Wang, Co-chair
Godmar V. Back, Co-chair
Danfeng (Daphne) Yao

May 7th, 2021
Blacksburg, Virginia

Keywords: Computer Systems, Containers, Container Security, Fuzzing, Fuzz Testing, Syzkaller, gVisor

TORPEDO: A Fuzzing Framework for Discovering Adversarial Container Workloads

Kenton R. McDonough

(ABSTRACT)

Over the last decade, container technology has fundamentally changed the landscape of commercial cloud computing services. In contrast to traditional VM technologies, containers theoretically provide the same process isolation guarantees with less overhead and additionally introduce finer grained options for resource allocation. Cloud providers have widely adopted container based architectures as the standard for multi-tenant hosting services and rely on underlying security guarantees to ensure that adversarial workloads cannot disrupt the activities of coresident containers on a given host. Unfortunately, recent work has shown that the isolation guarantees provided by containers are not absolute. Due to inconsistencies in the way cgroups have been added to the Linux kernel, there exist vulnerabilities that allow containerized processes to generate "out of band" workloads and negatively impact the performance of the entire host without being appropriately charged. Because of the relative complexity of the kernel, discovering these vulnerabilities through traditional static analysis tools may be very challenging. In this work, we present TORPEDO, a set of modifications to the SYZKALLER fuzzing framework that creates containerized workloads and searches for sequences of system calls that break process isolation boundaries. TORPEDO combines traditional code coverage feedback with resource utilization measurements to motivate the generation of "adversarial" programs based on user-defined criteria. Experiments conducted on the default docker runtime runC as well as the virtualized runtime gVisor independently

reconfirm several known vulnerabilities and discover interesting new results and bugs, giving us a promising framework to conduct more research.

TORPEDO: A Fuzzing Framework for Discovering Adversarial Container Workloads

Kenton R. McDonough

(GENERAL AUDIENCE ABSTRACT)

Over the last decade, container technology has fundamentally changed the landscape of commercial cloud computing services. By abstracting away many of the system details required to deploy software, developers can rapidly prototype, deploy, and take advantage of massive distributed frameworks when deploying new software products. These paradigms are supported with corresponding business models offered by cloud providers, who allocate space on powerful physical hardware among many potentially competing services. Unfortunately, recent work has shown that the isolation guarantees provided by containers are not absolute. Due to inconsistencies in the way containers have been implemented by the Linux kernel, there exist vulnerabilities that allow containerized programs to generate "out of band" workloads and negatively impact the performance of other containers. In general, these vulnerabilities are difficult to identify, but can be very severe. In this work, we present TORPEDO, a set of modifications to the SYZKALLER fuzzing framework that creates containerized workloads and searches for programs that negatively impact other containers. TORPEDO uses a novel technique that combines resource monitoring with code coverage approximations, and initial testing on common container software has revealed new interesting vulnerabilities and bugs.

Dedication

This work is dedicated to the many people that have supported me during my education. I would like to thank my family, and especially my parents, for encouraging me to pursue whatever goals I chose for myself while I was at school. I would like to thank my brother for reminding me to take time away from academics and pursue my passions outside of work. I would like to thank my friends for being the best support group anyone could ask for, especially during the darkest parts of the pandemic this past year. I would like to thank my coworkers and mentors at Viasat for their support and generosity while I have finished this part of my education. Finally, I would like to thank the entirety of the Marching Virginians for providing me with so many opportunities over my entire college career. I would be nowhere close to the person I am today without the constant support from our organization, and especially my family in the Trumpet section.

Acknowledgments

I would like to begin by thanking Dr. Wang for taking me on as a student and mentoring me through Graduate school. I appreciate how difficult it must be to approve an independent project while also maintaining a lab, but I appreciate his trust and support while I've worked this past year. Next, I would like to thank Dr. Xing Gao and Dr. Shuai Wang for their constant guidance and help on this project. I look forward to continuing work on this system and hopefully presenting our results at a larger forum this upcoming year. Finally, I would like to thank Dr. Back and Dr. Yao for their contributions to my education and professional development. Dr. Yao's cybersecurity research class was exceptionally valuable to my understanding of the state of the art in our industry, and the opportunity to work and continue learning with Dr. Back while administering CS3214 has been one of the most valuable experiences I will take from my college career.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Container Fuzzing

Fuzz testing [43] has long been hailed as an efficient method of hunting for bugs and exposing unusual behavior in a program. By combining feedback mechanisms with a smart input-generation algorithm, automated fuzz testers can effectively test hard-to-reach code blocks and identify interesting edge cases for human testers to evaluate. Traditional static analysis tools can often identify simple bugs and common error conditions, but for extremely large or complex code bases, fuzz testing is more efficient at isolating complex interactions that result in unwanted behaviors.

Container systems represent a complex intersection between userspace software, kernel APIs, and user supplied workloads. As such, verifying that these systems are functioning correctly for all supported workloads is a non-trivial task. Recent work has shown that there exist "adversarial" workloads which can circumvent container isolation mechanisms and degrade performance, leak information, and otherwise negatively impact workloads of other tenants. [21], [20]. So far, these examples have been identified using manual static analysis techniques, but due to the complexity of the systems involved, we believe that fuzz testing would be an effective technique for exposing such "adversarial" behavior.

To the best of our knowledge, no current fuzz tester is capable of dynamically generating

workloads to run in container systems. However, because container systems are largely implemented as augmentations to existing OS process abstractions, we consider an operating system fuzzer to be an appropriate starting point. In particular, we find SYZKALLER [58], an OS fuzzer from Google, to be a promising framework to extend for our work.

1.2 Proposed Solution

We propose TORPEDO, a set of modifications and additions to the SYZKALLER framework to create containerized workloads, observe resource utilization, and combine the former with code coverage feedback to hunt for adversarial sequences of system calls. TORPEDO augments the existing functionality of SYZKALLER in the following ways:

1. **Adding In-Container Fuzzing**. Currently, SYZKALLER is designed to work with OS kernels loaded into virtual machines. We modify the VM translation layer to add support for Docker, the most common API interface for interacting with containerized processes [56].

2. **Capturing Resource Utilization**. To identify adversarial workloads, we implement a series of fine-grained resource tracking mechanisms into SYZKALLER. We primarily focus on CPU utilization measurements, but anticipate this could be extended to other relevant metrics in the future.

3. **Extending the Fuzzing Algorithm**. By combining feedback from resource utilization measurements with existing code coverage measurement from SYZKALLER, we augment the existing generational algorithm to select programs that exhibit adversarial behaviors. We introduce a secondary state machine that operates on a set of ‘n‘ programs to guide decision making and ensure measurement accuracy.

4. **Adding Seed Ingestion and Minimization**. To improve test reproducibility, we introduce a Syzkaller workflow for ingesting seed programs. We additionally provide a workflow for minimizing adversarial programs in the context of resource observations.

5. **Retaining Scalability** We design Torpedo to operate irrespective of the backing runtime for the container system. To retain Syzkaller's inherent efficiency, we introduce a series of synchronization mechanisms that allow for multiple fuzzing processes to run simultaneously without compromising measurement accuracy.

1.3 Contributions

This research introduces a novel fuzzing technique suitable for evaluating container workloads in the context of a multi-tenant cloud. We make the following contributions:

1. **A new approach to guide fuzzing**. We design a generational fuzzing algorithm that combines resource utilization observations with traditional code coverage mechanism to guide the generation of adversarial workloads.

2. **Working implementation**. We implement a series of patches and architectural changes to the well known Syzkaller OS fuzzer to support Torpedo while preserving the underlying crash detection facilities.

3. **Realistic Evaluation**. We test Torpedo using hundreds of high quality seeds that emulate the behavior of real programs. We test two different container runtimes that utilize distinctly different designs.

4. **Legitimate Results**. We examine several adversarial patterns identified by Torpedo, including some that to the best of our knowledge have not been described in any

literature.

1.4 Roadmap

In Chapter 2, we discuss the implementation and history of container systems and different kinds of fuzzers. We also review several documented container vulnerabilities and attacks on cloud container systems. Finally, we describe the architecture of SYZKALLER in detail.

Chapter 3 discusses the algorithms we conceive to address combining multiple feedback mechanisms and how we propose to identify a workload as adversarial. We explain how SYZKALLER was modified to support these changes from a high level.

Chapter 4 details the evaluation procedure for TORPEDO, including the seed selection strategy and operational parameters for our algorithm. We discuss several interesting results identified during our fuzzing campaign and the implications these have for containerized workloads. Supporting code snippets and utilization tables are referenced from appendix A

Finally, Chapter 5 and Chapter 6 present some discussion about the quality of the results and notes for improvement, followed by a summary of our efforts.

Chapter 2

Review of Literature

2.1 Virtualization Techniques

For many years, Virtual Machines have been the standard unit of isolation for efficiently managing and sharing computing resources. The ability to run multiple virtualized operating systems inside a single host has a variety of applications spanning several computing disciplines. The rise of multi-tenant cloud computing, however, highlighted several limitations of VM technologies and motivated research into alternative software isolation techniques.

Primarily, the limiting factor on the use of VMs is their overhead. A recent study performed by Stratoscale [9] noted the overhead of running a virtual machine manager (VMM) was approximately 10%, meaning that any guests running on that host would be competing for only 90% of the CPU. Simple benchmarks on CPU and I/O performance showed a latency between 25% and 30% between processes running in a VM and running on bare metal. While it could be argued that the growth of containers has stunted VM research over the past decade, this test still demonstrates the significant performance losses associated with VMs. No matter how the technology advances, there will still be significant drawbacks to hosting multiple copies of an operating system in several instances, rather than sharing one host operating system on bare metal.

The one area where VMs reign supreme is isolation and security. By running an entirely

sandboxed operating system, it is impossible for a process inside a VM to escape onto the host and directly execute instructions. For many security-critical applications, using a VM is the only technique that appropriately isolates the processes and prevents them from being tampered with. Container technology has largely evolved to simulate the isolation guarantees of VMs using operating system features, but for various reasons, the implementation of such mechanisms is still incomplete.

2.2 Containers

Fundamentally, a container is an OS level abstraction describing a set of segregated resources. Processes running in a container should be isolated from all other processes on the host and be collectively constrained by the resources assigned to that container by the host. Despite this description, containers are not first class objects. They are an abstract concept implemented by various kernel systems to create the desired effect. In general, a modern container is defined by three kinds of constraints: resources, visibility, and access.

The notion of what constitutes a "Container" has changed several times since the concept was first introduced. One of the first usages of the term "Container" is by Banga et al. in their 1999 OSDI Paper *"Resource Containers: a new facility for resource management in server systems"* [6]. This early notion of a container focuses on the division of resources among threads but not visibility control or access control. Around the same time, Kamp et al. conceived of a similar "security container" (called a 'Jail' in FreeBSD terminology) which imposed access restrictions onto arbitrary untrusted user processes [25]. These two concepts received much attention over the following years, but weren't formally brought together until 2007, when Soltesz et al. combined several concrete Linux subsystems to create the first "container based OS" [52] and cement the modern notion of a container into literature.

In 2006, Google revealed a suite of patches to kernel resource tracking mechanisms called "Process Containers" [11]. The resulting system was renamed *Control Groups* to avoid confusion with the existing "Container" terminology, and introduced an implementation of hierarchical process groups with associated "controller" subsystems to enforce resource constraints. Linux kernel version 2.6.24 was the first mainline release to feature support for cgroups, which form the basis for controlling container resource usage.

Orthogonal development on visibility restriction via namespaces began as far back as 2002 (Kernel version 2.4.19) and continues today. The ability to sandbox one or more processes with respect to a kernel system is very useful for testing new features or as part of hardware emulation in addition to the obvious security benefits. Namespaces are essential to the visibility restrictions that secure containers.

Mandatory Access Control (MAC) modules, which in turn rely on the Linux Security Modules framework, are a way of preventing unauthorized access to data. Unlike standard Discretionary Access Control (DAC), wherein the owner of a resource can define and mutate access permissions, MAC policies are created by a trusted security administrator and cannot be changed or circumvented by system users. These systems are commonly used to enforce disk access restrictions for containers, which could attempt to read or write resources outside the scope of the container itself.

In addition to the above, containers are also scrutinized with respect to their access to the kernel. In particular, the set of system calls a containerized process can make, as well as arguments to those calls, is often restricted. This is done through use of the Secure Computing (seccomp) interface [15], which was merged into the kernel in version 3.12. Among other things, these restrictions can prevent a compromised application from exploiting vulnerabilities in the kernel by preventing "dangerous syscalls" from happening in the first place.

cgroup	Description
cpu	Minimum and Maximum CPU shares
cpuset	Physical Cores available for scheduling
memory	Upper limit on process, kernel, and swap memory
net_prio	Priority for outgoing network traffic
blkio	Bandwith for communication with block devices

Table 2.1: common cgroups used for containers

2.2.1 cgroups

From the manual page, a Control Group is "a collection of processes that are bound to a set of limits or parameters defined via the cgroup filesystem." [35]. Each limit must be enforced by a *Subsystem*, also known as a *Resource Controller* or simply a *Controller*. Controllers are responsible for modifying the behavior of a process to correspond to the limitations imposed by a cgroup; for example, the CPU controller would prevent a process from receiving a larger share of the CPU than its cgroup allows. A summary of commonly used cgroups and their description is provided in Table 2.1.

By design, cgroups are inherited by child processes as a result of fork(2).

To support a hierarchical structure, all processes are intrinsically part of a root cgroup which defines no restrictions. Additional cgroups are arranged beneath this root in a tree, with restrictions from parent cgroups carrying through to lower groups. Internally, this hierarchy is maintained using the cgroupfs pseudo-filesystem. cgroup usage information is exposed by the kernel and can be queried using system utilities like *systemd-cgtop* [51]. Certain Container Engine components (Section 1.2) will also expose cgroup information for general use.

2.2.2 Namespaces

From the manual page, a Namespace "wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource." [36].

For containerized applications, several namespaces are combined to give processes the illusion that they are running on an entire host. The *PID* namespace is employed to restrict the visibility of processes outside the container. The *Network* namespace provides an entirely different set of networking resources for processes, including IP addresses, routing tables, and port numbers.

In particular, the introduction of the *User* namespace in 2013 (kernel version 3.8) [34] made the current notion of containers as no different than a lightweight VM attainable. The User namespace provides the illusion that the first container process has UID 0 (is root), although namespace translation techniques like subuid [37] are necessary to prevent a privilege escalation attack [31] (see Section 2.4.2).

2.2.3 Linux Security Modules (LSMs)

In general, Linux Security Modules are meant to provide another layer of isolation for virtualized hosts in multi-tenant environments. By default, containers have access to a number of common system utilities and can potentially access areas of the disk segregated to another container or the host. By adding an enforcement policy, containerized processes can be constrained using an explicit allow-list that specifies which areas of the disk are within limits.

Two common examples of production Linux security modules are SELinux [24] and AppAr-

mor [10]. SELinux is a Red Hat open source project that was merged into the Linux Kernel in 2003 (version 2.6.0). The module works by assigning each file on disk and each process a label. When a process makes a request to access a resource, the module makes a decision as to whether the process type is allowed to access the resource type, and either allows or denies the request. For convenience, SELinux explicitly defines some labels for containerized processes [48].

AppArmor is a similar LSM developed by Canonical. AppArmor deals in explicit policy documents called ”profiles” which specify rules that should be applied to a process. These rules can include allow and deny lists for file paths, system calls, Linux capabilities, network devices, and more. Each policy is parsed into the kernel module and applied to matching processes when they request access to resources; Docker (section 2.3.1) includes built-in support for adding an AppArmor profile to a container on the command line [28].

2.2.4 Seccomp Profiles

Seccomp denylists, which are often referred to as ”profiles”, use a generalized form of the Berkeley Packet Filter language. Each profile is loaded into the kernel and checked at call time; depending on the enforcement mode, the calling process could be warned or terminated for attempting an action outside the filter restrictions. Docker enforces a default seccomp profile by default and provides a command line parameter for a custom profile to be passed from userspace [30]. These profiles are often computed by testing the application that will be containerized and recording all required operational syscalls, then explicitly denying everything else.

2.3 Container Engines

In order to create a container, several kernel systems must be manipulated by a higher level piece of software to fork "containerized" processes. Ensuring that these processes also have access to segregated directory space, runtime libraries, and environment variables is also non-trivially complex. To make this process easier and provide a system-agnostic API, developers have created various "Container Engines" to create and manage the lifecycle of containers. Two common examples of container engines are Docker [27] and Podman [23].

Docker was the first major container engine to be developed, and while it has changed significantly since its inception, is understood to be the defacto industry standard. In 2018, Docker reported that over 37 billion containerized applications have been downloaded through Docker operated repositories [56]. Estimates placed in 2016 predicted that Docker's revenue would grow to over 3.4 billion dollars by 2021, and the software has been adopted by all major cloud providers as a basis for their container services. For these reasons, this work will focus primarily on Docker's architecture.

2.3.1 Docker

Docker employs a client-server architecture to create and manage containers. Docker clients communicate with the Docker Daemon over HTTP to create, manage, and destroy containers. The Docker Daemon maintains a registry of container images cached from an upstream repository and maintains information about the state of each container. Docker's official diagram for this architecture can be found in Figure 2.1 [29]

To actually create containers, the docker daemon uses a series of helper binaries and libraries. Each binary is responsible for one specific component of Docker's overall functionality, and

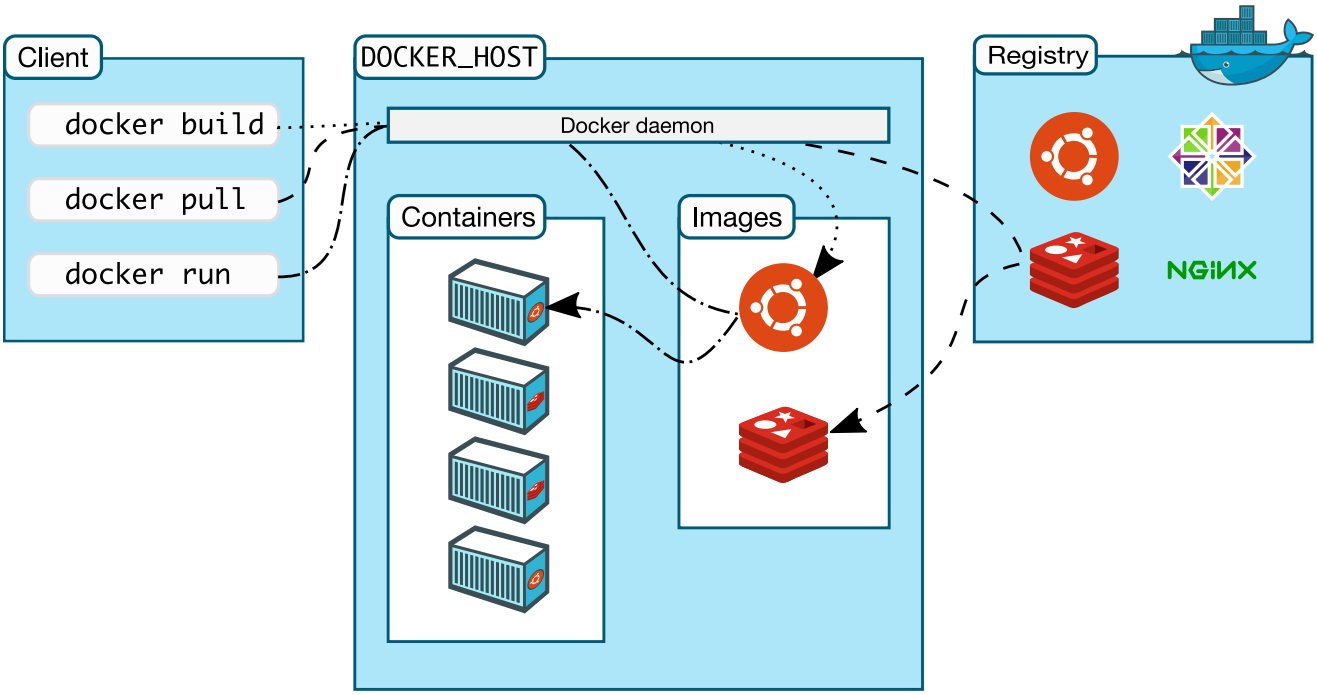

Figure 2.1: Overview of Docker Architecture

are shown in Figure 2.2 [2]

containerd

As its name suggests, containerd is responsible for managing the first-class container objects exposed by the docker API. The documentation for containerd helps describe what functionality it provides, and is duplicated in Table 2.2 [12]

containerd-shim

containerd-shim is a utility process that servers several purposes. Primarily, starts and allows the container runtime (Section 2.3.2) to exit after the container process has been created [26]. It also allows IO pipes for the container process to remain open in the event dockerd terminates and cleanly communicates the container's exit status without forcing the

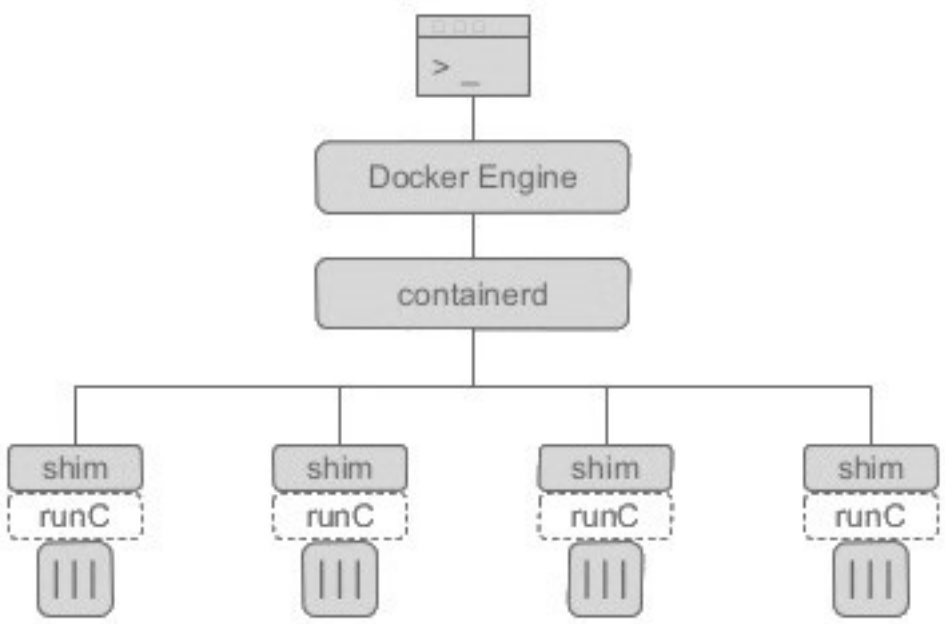

Figure 2.2: Docker Internals: containerd and runC

Feature	In/Out	Description
execution	in	create, start, stop, resume, signal, and delete containers
distribution	in	push/pull images, as well as operate on image objects
metrics	in	container-level metrics, cgroup stats and OOM events
networking	out	should be provided via higher-level systems
build	out	containerd operates on images, but doesn't create them
logging	out	stdout streams are passed to higher level systems

Table 2.2: containerd functionality

parent to explicitly wait.

runC

runC is the default container runtime (See Section 2.3.2) packaged with Docker. It interfaces with the kernel, forks the container "entrypoint" process, applies resource restrictions and terminates.

2.3.2 Container Runtimes

From a security perspective, the Container Runtime is the most critical component in the container setup process. This binary is responsible for translating the visibility and resource restrictions from the user-facing API into directives for the kernel. For a typical container this involves the creation of a new set of namespaces, one or more cgroups to restrict resource usage, and access policies to govern allowed file operations and system calls.

There are three major existing designs for container runtimes: "Native" runtimes, "Sandboxed" runtimes, and "Virtualized" runtimes [5]. Native runtimes perform the necessary setup for the container and then exit, allowing the container to completely share the host kernel with the containerized process. As the default runtime packaged with Docker, runC is the most common example of a native runtime. Sandboxed runtimes introduce a translation layer between the container and the host kernel by means of a kernel proxy. gVisor [54] is a popular example of 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. Since gVisor does not support the entire Linux system call interface, however, not all applications are supported. Finally, virtualized runtimes use an entire virtual machine to house containerized processes and isolate them from the host. The Kata project [19] is the defacto

standard for this style of runtime, which uses a special VM hypervisor to proxy information in and out of the containers. The use of full virtual machines does introduce a nontrival performance overhead, although projects like AWS Firecracker [1] have reduced this overhead to acceptable levels.

2.3.3 Container Orchestration Systems

Container Engines are primarily concerned with creating containers that match a developer-provided specification. How these containers are arranged to meet the needs of a specific piece of software or business, however, is an orthogonal issue solved by a Container Orchestration System. Container Orchestration Systems are responsible for tasks like replicating containers, load-balancing between networked containers, mounting storage volumes, and collecting logs [16].

Kubernetes

The "Gold Standard" in container orchestration is Kubernetes [4]. Kubernetes groups containers into units called "pods" and has a mature API to manipulate them. Kubernetes currently implements container functionality on top of Docker components, although the Kubernetes authors have plans to sunset the docker runtime in favor of a new, CRI-compatible runtime in the near future [3]. Other players in the Container Orchestration field are Docker Swarm and Apache Mesos [18], although Kubernetes has emerged as a clear leader.

2.4 Understanding Container Vulnerabilities

Because containers are implemented using a combination of semi-unrelated kernel systems, we may reasonably suspect there to be gaps in the isolation they provide. In fact, several high profile papers have been published in the last few years detailing serious vulnerabilities in container security mechanisms and potentially dangerous exploits the former could support.

2.4.1 Namespace Visibility Vulnerabilities

As discussed in section 2.2.2, Namespaces are meant to prevent containerized processes from obtaining information about other objects and activities occurring on the host system. If properly implemented, a containerized process should be unable to determine any information about other processes, if any, running on the same host.

To determine why hiding this kind of information is useful, we must consider the types of applications and business that container systems often support. Containers are primarily used to package software that should be environment agnostic and can be run on any system that supports a container engine. As long as the container is assigned the same resources, the application should notice no difference in the quality of service it receives. Cloud providers take advantage of this to implement high-efficiency load balancing algorithms that distribute containerized loads over extremely large pools of physical hardware with no concern for the actual contents of the container.

Even though namespaces can effectively create isolation at the software level, certain observable fluctuations in the physical hardware supporting the process are impossible to eliminate. For example, cache-hit timing statistics are subject to manipulation by all processes running on the host and are often used to construct side-channel attacks, such as the widely researched

Spectre exploit in 2018 [45]. Such cache-based side channel attacks are widely researched and in some cases have been shown to accurately infer secret data like cryptographic keys.

Naturally, these inference techniques rely on the malicious process being on the same physical host as the victim process to have access to the shared cache. Due to the aforementioned load balancing strategies employed by cloud providers, there exist strategies to map and strategically deploy instances until coresidence with a certain target instance is achieved [50]. Ristenpart Et. Al. show how coresidence can be determined easily using a naive network implementation or more situationally by examining cache load. Although the techniques described by primarily deal with VMs, the same techniques are generally applicable to containers.

Rather than attempting to infer coresidence with an arbitrary victim process, an attacker may wish to infer coresidence between two malicious containers under their control. To this effect, there are several documented of examples of certain kernel features that are not yet namespaced and may leak information to namespaced processes. Gao Et. Al. investigated a number of information leakage channels in kernel pseudo-filesystems and demonstrated how these channels could be used to infer coresidency between two malicious containers on the same physical host [20]. Furthermore, they hypothesize how this technique could be used to cheaply implement a synergistic power attack in a data center. Kedrowitsch et al. discuss a real case of malware exploiting a namespacing vulnerability to discern whether the execution environment is containerized [33], which unfortunately lessens container application as honeypots.

By exploiting vulnerabilities in kernel namespacing features, we see that it is possible to infer coresidence between two containers on a commercial cloud system and use this information to mount attacks against the provider or a particular victim container.

2.4.2 Privilege Escalation

By default, PID translation between a container and the host is 1:1. In other words, UID 0 (root) inside the container corresponds to UID 0 on the host. Depending on the level of access given to the container, this makes it trivially easy to impersonate root (or another known user) and make changes to various interfaces. Since many programs rely on the perception of being root to conduct operations, container engines prefer to solve this privilege problem by anonymously translating a namespaced root UID into an unprivileged anonymous user on the host. This is done using the subuid/subgid system [37], which provides a mechanism to translate namespaced PIDs into unprivileged "machine" users with high UIDs on the host. However, this technique is not enough to stop exploits of kernel vulnerabilities that can lead to privilege escalation. Such attacks are called "container escapes" [41] and frequently result in an attacker opening a root shell on the host, from which they can arbitrarily affect other processes and steal data. These vulnerabilities can also arise from the implementation of a container runtime, discussed in Section 2.3.2.

2.4.3 cgroup Vulnerabilities

As discussed in Section 2.2.1, cgroups are designed to impose absolute limitations on the resources a process may consume. While a variety of controllers exist for different subsystems, the most common control access to physical cores, CPU cycles, virtual memory, and block IO devices.

In essence, cgroup controllers perform two important tasks. The first is to accurately track information associated with the specific cgroup, and the second is to use this information to make decisions about whether a process should be allowed to continue some activity. For example, the CPU controller needs to accurately track of how much CPU the cgroup

has used and, if the cgroup quota is exhausted, not allow the cgroup to become scheduled until the quota has refreshed. Cloud providers are particularly interested in both of these functions for different reasons. The tracking functions provided by cgroups allow them to implement fine-grained "pay for what you use" business models to entice customers, and the limitation mechanisms make it trivial to partition powerful hardware into performance units and provide customers with quality of service guarantees.

In general, the systems that govern cgroup scheduling decisions are understood to be complete. Several issues arise in the implementation of resource tracking mechanisms, however, that can allow processes to 'escape' their cgroup boundaries and consume additional resources. We refer to such processes as creating "Out of Bounds" workloads that are not accounted for by the kernel. Gao. Et. Al. [21] published a comprehensive study of several cgroup-escape techniques for various kernel subsystems in 2019. At the heart of these techniques is the idea of "work deferral" by causing another process (in a different cgroup) to do work on behalf of a constrained original process. The different generalizations of this technique are described below.

Deferring work to the kernel

To ensure work is completed in a timely manner, the kernel has many interfaces for scheduling tasks and completing them asynchronously. Typically, these tasks are executed by one or more "kworker" threads which are initially forked from the kernel thread daemon "kthreadd". For obvious reasons, all kernel threads are assigned to the implicit root cgroup that defines no restrictions on resource consumption.

If a userspace process is capable of manipulating a kernel work queue and waking a kworker thread to execute a task, this process has effectively delegated some amount of work to a

cgroup beside its own and created an "out of band" workload. One easy example of this deferral explored in [21] is the creation of coredumps by the kernel exception handling mechanism. When a userspace process receives a signal that causes it to terminate, the kernel creates a new kworker thread and causes it to execute a registered userspace utility to generate the coredump by means of the *usermodehelper* API. Using this method, containers can cause up to 200x more CPU utilization than the cgroup controller reports by simply generating exceptions.

Another example from [21] involves use of the sync [38] family of system calls. As part of a sync operation, the kernel tasks one or more kworker threads with flushing some amount of cached data to the physical disk. In addition to creating out-of-band CPU utilization, this method has the additional side-effect of forcing all processes performing IO on the system to wait while the synchronization is complete. Additionally, due to limitations in the blkio cgroup, the process that called sync is *not* charged for any disk operations.

Deferring work to other process cgroups

Similar to how kernel threads can be delegated work by userspace processes, various system daemons and services in other cgroups can be forced to do work on behalf of a containerized process.

One standard example of this arrangement is the kernel audit subsystem, which consists of three processes: kauditd, auditd, and journald. The two former audit daemons are responsible for collecting and registering certain system events, like the use of the "sudo" command. The latter journal daemon is responsible for writing audit events into a log where they can be offloaded or parsed. All three daemons run in either the root cgroup or a dedicated cgroup unrelated to userspace processes.

Because the kernel is unaware of which processes are containerized, it has no way of distinguishing between events that occur in a container and events that occur on the host. As such, the audit daemons and the journal daemon perform work on behalf of all containerized processes, but this resource consumption is not tracked to the originators. A malicious process that triggers many audit events can easily exploit this to degrade the performance of all processes on the host.

Another exploitable mechanism for delegating work comes in the form of kernel interrupt handling mechanisms. By design, interrupts are serviced in the context of whatever process happened to be running when the interrupt was received. Hard IRQs are performed outside of process context, so no resource consumption can be tracked whatsoever, but corresponding softirqs are handled in the context of whatever unlucky process is returning from a hard IRQ. In the general case, this means that processes outside the cgroup of the originator may be charged for resources consumed executing a softirq on behalf of some other process[1]

Mitigation techniques

Some work has already been done to appropriately track out-of-band workloads for containerized processes. In particular, the overhead from processing network traffic in interrupt context can significantly decrease the amount of CPU available for other containers on a host. Khalid et al. propose a solution to this problem by introducing the concept of a "credit" based scheduler in IRON [40]. IRON is capable of uniquely associating a network packet with an originating process and tracking the amount of time an unlucky process in interrupt context may spend processing that packet. cgroups that create an excess of traffic are penalized in the next time slice, and the unrelated processes that handled the packets

[1]If the amount of softirqs becomes too large, a dedicated ksoftirq thread is woken to execute softirqs in process context. This thread is also forked from the kernel thread daemon, and so work is charged to the root cgroup rather than the originating process.

are given a bonus. In CFS terminology, this does imply the per-process vruntime be allowed to become negative in the event a heavy penalty was applied, effectively starving the process of CPU time and preventing it from creating additional packets.

In some cases, the kernel could be modified to assign kworker threads into the cgroup of the originator for a given work item for tracking purposes. The *usermodehelper* API vulnerability in particular can be patched with an extremely minor change to one kernel module and its caller.[2].

2.5 Fuzzing

Fuzz testing (fuzzing) originated in the 1990s as a class project to find bugs in Unix command line utilities [43]. The idea has proven successful enough to apply to practically every field of application in computing, and due to the automated nature of fuzzing algorithms, is typically very efficient.

During fuzz testing, an application is selected and the ways in which it can accept input are enumerated. For traditional command line utilities, this may be data passed via STDIN or read from a file. For graphical utilities, this may be a combination of mouse clicks and keyboard presses, including text entry. For web based applications, this may be a set of URLs or cookie values sent to a server. For traditional operating systems, this would be a selection of system calls and arguments.

Based on the manner of input, a fuzzing program is constructed to generate inputs and monitor the application state after the input is received. Under most circumstances this means checking for a crash, although more modern fuzzers are capable of monitoring for

[2]I helped implement and test this patch for CS5264 Linux Kernel Programming project at Virginia Tech in Fall 2020. The code for the patch is available at https://github.com/kent007/lkp-5.8

other heuristics, like algorithmic complexity vulnerabilities (See Section 2.5.5). The fuzzer bombards the program under test with inputs for some interval of time and records what inputs trigger abnormal behavior, which humans may manually confirm and isolate to aid in debugging.

2.5.1 Awareness of Program Structure

The manner in which a fuzzer generates inputs varies greatly depending on the class of application under test and the amount of knowledge the fuzzer has about the application. Early fuzzers primarily focused on completely random inputs, which will eventually exhaust all possible inputs but likely not within a reasonable finite time interval [8]. To increase efficiency, fuzzers typically incorporate some kind of information or feedback from the program under test to drive generation of more targeted inputs. In general, there are 3 classifications of fuzzer based on the amount of awareness it has about the structure of the application

- *Black-Box fuzzers* treat the application under test as a "black box". The fuzzer has no information about the internal structure, and may only learn by observing the output, if any, from various inputs.
- *White-Box fuzzers* use static analysis techniques to understand the various paths input data may take in an application and attempts to create inputs that trigger critical code sections.
- *Grey-Box fuzzers* use lightweight instrumentation to track program coverage for each input without requiring extensive knowledge of the application under fuzz.

Black-box fuzzers are often incapable of exposing more than "shallow" bugs due to the relative unlikelihood that randomly generated input will trigger specific behavior in a given

application. Similarly, white-box fuzzers are often impractical if the source code for the application isn't available and can take significant time to trace and generate an input. In practice, grey-box fuzzers obtain the benefits of both techniques and are suitable for most types of fuzzing.

2.5.2 Input Generation Technique

Regardless of input structure, fuzzers must be capable of creating new inputs for the application under test. The simplest technique for doing so is by simply generating new ones from scratch. This technique results in a high volume of inputs, but potentially may waste time testing uninteresting parts of the program. A potentially more lucrative technique is "mutational" fuzzing, which uses a genetic algorithm [44] on a corpus of seed programs to achieve high coverage.

Genetic algorithms begin with a population of known valid inputs, also called "seeds". The algorithm additional defines one or more operators that mimic phenomena observed in biological systems, such as "mutation" and "crossover". The implementation of these operators may vary greatly depending on the structure of the input seeds.

During each round of testing, the fuzzer selects some number of seed programs and applies one or more biological operators to produce new, although similar, inputs. These inputs are then tested and some quantifiable result is recorded. Finally, the algorithm goes through a "selection" process to discard any inputs from the new generation that failed to produce new results than the parent generation. The remaining inputs are incorporated into the corpus, and a new generation is prepared.

To achieve the best results, some fuzzers are capable of generating inputs and implementing a generation algorithm when a suitable number of interesting seeds has been identified.

2.5.3 Awareness of Input Structure

Some applications operate on input matching a specific protocol, file format, or formal grammar. Fuzzers can largely be categorized into "smart" fuzzers that respect input format when constructing inputs, and "dumb" fuzzers that do not.

"Smart" fuzzers must be supplied with the formal definition for program inputs as well as code to construct inputs. This technique functions well when the input format is simple, well-known, or otherwise easy to manipulate.

"Dumb" fuzzers construct or mutate blocks of input without necessarily respecting the constraints of the input format. In general, this results in more efficient input generation, but many of these inputs may not be valid or significant. One common example where "dumb" fuzzers are ineffective is any input format that contains a CRC [59]. To actually fuzz application internals, the fuzzer *must* respect the input structure and compute a valid CRC for each input.

2.5.4 Existing Fuzzers

Fuzzers now exist for most classes of application, either as part of generic libraries or as dedicated projects. Fuzzing algorithms have improved immensely as more work has been done on the subject, and some fuzzers in particular stand out above the rest.

AFL

AFL is a "brute-force fuzzer coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm." [22]. AFL uses a form of Gray-box instrumentation to collect basic coverage information, but can also function appropriately in black-box mode when the bi-

nary cannot be instrumented. Due to its relative success at fuzzing many kinds of programs, AFL is very popular and has a number of derivative projects [60].

AFL requires a corpus of known good test files. These files are first minimized to isolate the smallest subset of input that creates the identified coverage profile, then mutated using a number of common techniques like adding, removing, or flipping bits. The resulting inputs are rerun and, should they generate new input, added into a test queue to repeat the process.

AFL is the defacto standard for most fuzzers with respect to features like speed, accuracy, and parallelization.

LibFuzz

LibFuzz is a "in-process, coverage-guided, evolutionary fuzzing engine" integrated into the LLVM compiler ecosystem [49]. LibFuzz operates on one or more functions from a target library directly by calling them directly with mutated input files. During the compilation process, LibFuzzer instruments the code with LLVM SanitizerCoverage and links against the libfuzzer library to produce a standalone fuzzing binary with the target function as an entrypoint.

LibFuzzer is similar to AFL in many ways and even has a switch to enable strict compatibility mode with the latter. The resulting fuzzer binaries must be supplied with seeds, although inputs can also be generated on the fly. Because LibFuzz operates at the function level, testers have much more flexibility with selecting what functions should be fuzzed and what can be safely ignored. This can be very powerful depending on the use case, but is not a complete replacement for grey-box fuzzers like AFL, which are suitable for general use.

Syzkaller

Syzkaller is a "unsupervised coverage-guided kernel fuzzer" developed by Google. Unlike the above fuzzers, Syzkaller is a special-purpose fuzzer designed to fuzz operating system kernels through the system call interface. Each supported kernel under fuzz is booted into a VM and sequences of system calls (or simply 'programs') are passed into a special process running in the VM to be executed. In the event of a crash, Syzkaller collects relevant output from the VM log to discover what program triggered the crash, then attempts to minimize and isolate the bug for future evaluation. Compiled programs are saved into a corpus for later use in mutation algorithms.

Syzkaller collects coverage information from inside the kernel itself to guide the mutation of new inputs. This feature requires the kernel to be compiled with coverage support, a feature which was introduced into the kernel by the Syzkaller authors in 2016 [57]. Coverage information is exposed on a file in the debugfs system where one or more threads may collect coverage information for a sequence of calls. In the event coverage information is not available from the kernel, Syzkaller default to a weak feedback signal generated from the unique XOR of the syscall number and the return code.

An in-depth discussion of the Syzkaller design is given in Section 2.6.

2.5.5 Fuzzing with non-binary feedback

While the majority of the work surrounding fuzzing is related to isolating crashes, fuzz testing can also be used to find other kinds of interesting edge-cases and bugs that do not result in a crash. AFL lists a number of potentially interesting sanity checks (Section 11 in [22]) that could be used to identify interesting behavior, but requires that the source code be available and properly notated. This technique requires that developers expend time

manually specifying some number of binary conditions that constitute a failure similarly to how unit tests are written. While undeniably useful, this approach is not appropriate for many circumstances.

Traditional fuzz testing also has a shortcoming in that transitive program states are typically not quantifiable. Most fuzzers are only concerned with the mapping between an input and an output state; for these fuzzers, the only two valid outputs are ”crashed” or ”not crashed”. With proper support as described above, the fuzzer may be able to detect certain error conditions that do not explicitly trigger a crash. This perspective, however, fails to take into account the wide range of behaviors that may be exhibited by a program as the result of an input that do not result in errors or crashes. These behaviors might include abnormal spikes in resource utilization, edge-cases in time complexity, or for security sensitive applications, inadvertent privilege escalation or confused deputy vulnerabilities.

As an example, we consider a program that involves a so-called ”super-linear” regex [14]. Under typical fuzz testing, a super-linear regex may potentially go unnoticed if the fuzzer was not tracking the execution time and looking for discrepancies between different inputs. Depending on the particular heuristic, it may be difficult to identify whether a behavior exhibited by the program is ”abnormal” in the first place. Clearly, most existing fuzzers would be unsuitable for detecting these kinds of bugs, as they must be capable of incorporating non-binary feedback into the fuzzing process.

One recent fuzzer that incorporates non-binary feedback into its algorithm is HotFuzz [7]. This fuzzer is designed to search for algorithmic complexity vulnerabilities (specifically related to execution time) as a means of mitigating potential Denial of Service attacks. The fuzzer combines coverage information with execution time observations to selectively mutate inputs closer to worst case conditions. While this fuzzer is specifically targeted toward Java code running on an instrumented VM, the techniques it uses are generally applicable and

are adapted as part of this work.

2.6 Syzkaller

Due to the sheer complexity involved in fuzzing operating system kernels in a reproducible manner, Syzkaller uses a sophisticated architecture and several supporting libraries to generate, transmit, fuzz, and log sequences of system calls (hereafter referred to as "programs").

As part of operation, Syzkaller uses 3 binaries: syz-manager, syz-fuzzer, and syz-executor. The former two binaries, as well as the majority of the library code, are implemented using Golang. The executor binary and some specific supporting libraries are implemented in C++. A diagram of Syzkaller's architecture is available in the documentation [58] and is reproduced in Figure 2.3

2.6.1 Supporting Libraries

In order to be aware of input structure, Syzkaller includes several libraries that define the syntax for each syscall on supported kernels. The library introduces an intermediate representation to simplify several aspects of the syscall boundary, such as passing pointers to regions of dynamic memory, saving results of calls for reuse, and differentiating between calls that use different protocols (such as an IPv4 socket being different than an IPv6 socket). The library also includes support for serializing a program to a file and later reading it back into memory.

To implement a genetic algorithm, Syzkaller has a library that defines several operations that may be performed on a program as a whole. The major operations are listed below.

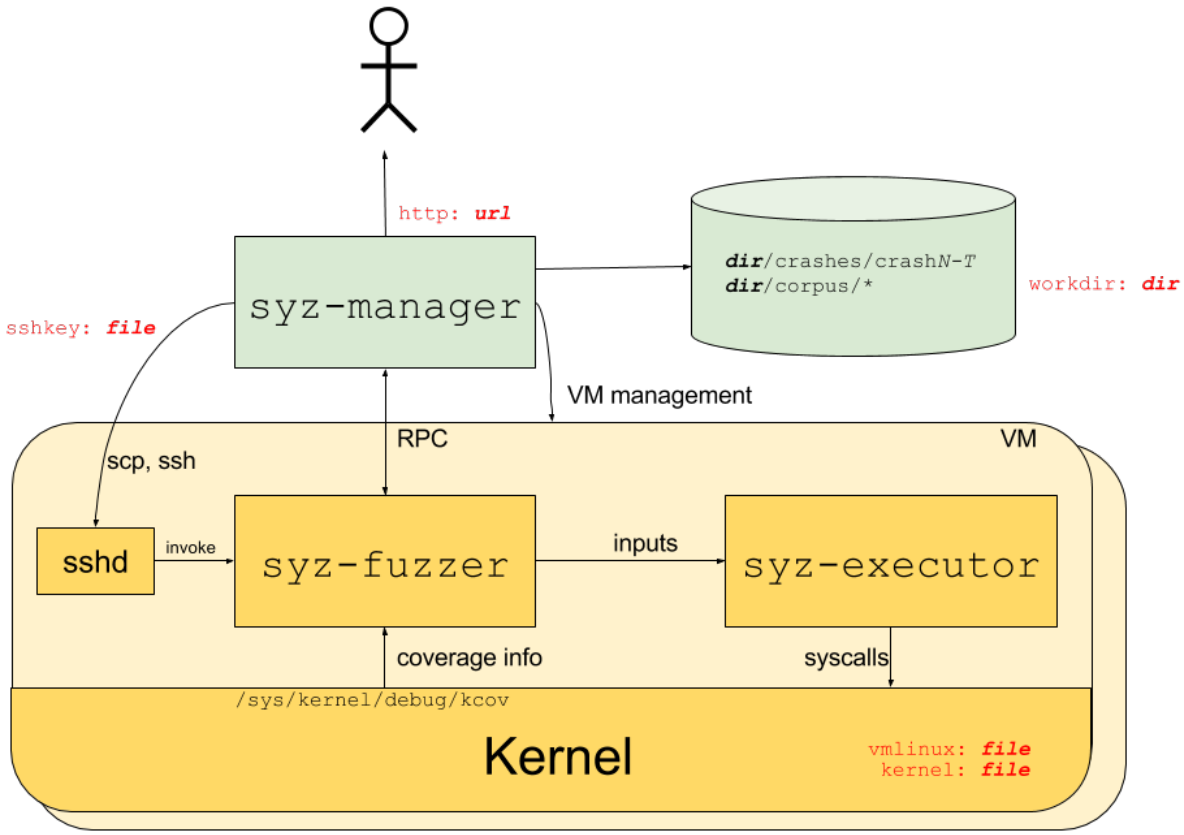

Figure 2.3: SYZKALLER Architecture [58]

1. Splice two programs. Syzkaller selects a random program from the corpus and chooses a certain number of sequential calls, then combines those with a randomly selected sequence of calls from the program being mutated. The resulting program is returned.

2. Add a call. Syzkaller computes a "bias" score across the syscalls already present in the program to select a syscall that is likely to interact with the calls already present. This operation is less likely when the program is at or near a max length.

3. Remove a call. Syzkaller deletes one call from the program. This operation is less likely when the program is very small.

4. Mutate a call. Syzkaller selects an argument of a random call and, using the semantics of the argument, randomizes the value. Certain preference is given to known interesting arguments like NULL or a bitfield of all 1s. Pointers to dynamic memory are typically filled with large random values.

2.6.2 Syz-Manager

The 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. During normal operation, each fuzzer routinely polls the manager to deliver operational statistics and potentially request new programs. The manager serves these statistics over a local HTTP server for human observers. When a fuzzer has exhausted a program, it contacts the manager and requests the program be added to the corpus. In this way, the manager serves as a synchronization mechanism for the corpus as well as a source of policy on what programs should be retained and which should be discarded.

Kernel	Notes
Linux	Complete
Akaros	Incomplete
Darwin/XNU	Requires OpenDarwin
FreeBSD	Mostly complete
Fuchsia	Complete
Gvisor	Complete, no coverage
Hafnium	Hypervisor; work in progress
NetBSD	Mostly complete
OpenBSD	Complete
Trusty	TEE; work in progress
Windows	Incomplete

Table 2.3: SYZKALLER Supported Kernels

Under normal operation, each fuzzer is assumed to run in a VM. The manager libraries include abstractions for working with a pool of VMs and include support for starting, logging, copying to, and stopping individual VMs. The full list of supported kernels is given in Table 2.3.

In the event a fuzzer suffers a crash, the manager attempts to reproduce and isolate the crash by reading through the fuzzer log data and isolating the last run program(s). The manager is not always successful in this regard, but can sometimes reproduce the crash down to a few lines of valid C code for manual inspection.

2.6.3 Syz-Fuzzer

The fuzzer binary is responsible for generating and manipulating programs through various lifetime stages. It multiplexes some number of asynchronous syz-executor binaries and transmits programs by means of shared memory or IPC pipes.

Each program moves through a simple state machine moderated by the fuzzer. A program begins in the ”candidate” stage, where it is run once to determine if it generates new coverage.

For each call in the program that generates new coverage, a triage item is created with a copy of the program. During the ”triage” stage, the program is rerun to verify it generates new coverage. It then moves immediately to the ”minimization” stage, where the fuzzer procedurally removes calls to identify the smallest version of the program that still produces the same new coverage. Finally, the program is moved to the ”smash” stage, where it is repeatedly mutated or injected with errors to determine variants that generate new coverage. These variants are reintroduced as candidates and the cycle continues.

To improve efficiency, all work items are placed into a synchronized work queue shared among all syz-executor processes. The queue itself prioritizes among the different type of work items; for example, a ”triage” item is more likely to be selected than a ”candidate” item if available.

Currently, Syzkaller does not have a facility for starting with seed programs. Each fuzzer starts by generating one or more programs from scratch and enqueueing them as candidates.

2.6.4 Syz-Executor

The executor is a C++ binary that reads in a serialized Syzkaller program and executes it while collecting coverage information about each call. To support exposing more interesting program behavior, the executor accepts a number of ”feature” switches (described in Table 2.4) to change the program environment or inject faults. By default, the executor enables a ”collider” mode that runs several syscalls simultaneously on different threads after a single-threaded run of the program.

If available, the executor collects kernel line coverage information from each call and returns the results of execution to the syz-fuzzer process.

Executor Switch	Description
SandboxSetUID	Impersonate Nobody User
SandboxNamespace	Execute code in a namespace
SandboxAndroid	Special android sandboxing
EnableTun	Use the /dev/tun interface for packet injection
EnableNetDev	Setup more network devices for testing
EnableNetReset	Reset the network namespace between programs
EnableCgroups	Setup cgroups for testing
EnableCloseFDs	Close FDs after each program
EnableDevlinkPCI	Setup devlink PCI device
EnableVhciInjection	Setup /dev/vhci for packet injection
EnableWifi	Setup mac80211_hwsim for wifi emulation

Table 2.4: Syz-Executor Additional Features

Chapter 3

Design and Implementation

To effectively fuzz containers, TORPEDO must satisfy the following requirements.

1. Deploy a set of containers configured with arbitrary constraints.
2. Distribute arbitrary fuzzing workloads to each container and synchronize their execution.
3. Collect code coverage and resource utilization feedback while containers are running.
4. Use both feedback mechanisms to drive program mutation.
5. Identify potentially adversarial workloads and flag these for evaluation.

We will address the design and implementation of each specific requirement in the later sections in this chapter. First, we consider the challenges in adapting Syzkaller to meet the needs of TORPEDO.

3.1 Considerations for using Syzkaller

SYZKALLER is designed to function by managing pools of virtual machines (see Figure 2.3). When choosing how to to adapt this architecture for TORPEDO, we made the following observations while experimenting with different options.

3.1.1 SYZKALLER's Reliance on Virtual Machines

SYZKALLER's use of virtual machines to perform fuzzing is an obvious choice. VMs allow for several instances of a kernel under test to be deployed in parallel, especially kernels that have been compiled with custom features or kernels designed to run on unavailable hardware. In particular, SYZKALLER's feedback mechanisms benefit immensely from getting access to kernel line coverage information, a feature which must be specifically enabled at kernel compilation time and is not included in most standard releases.

A large part of TORPEDO's novelty involves taking accurate resource utilization measurements and analyzing these to identify adversarial patterns exhibited by programs. To do so, TORPEDO will benefit by reducing the amount of resource overhead incurred by additional isolation mechanisms like VMs. Especially when considering sandboxed and virtualized runtimes, which need to be analyzed closely for adversarial utilization on the host, an additional layer of VM translation will make detangling measurements difficult and slow down the overall fuzzing process.

After some experimentation, we choose to run all SYZKALLER processes alongside one another on the same host. In this model, the syz-fuzzer process can monitor resource utilization for each executor individually and need not detangle any overhead from the use of an additional VM.

3.1.2 Kernel Code Coverage

Under SYZKALLER's original design, each executor runs inside a VM with code coverage enabled. These executors need only interact with the appropriate kernel pseudofile to collect high quality coverage information and use this as feedback to motivate program generation. Upon initial experimentation with different container runtimes, we discover that ‘gVisor‘

does *not* have support for the ‘ioctl(2)‘ necessary for coverage collection. In fact, Syzkaller contains sample configuration information for fuzzing gVisor outside of the container context that explicitly disables coverage collection.

As a fallback mechanism, Syzkaller computes a ”coverage” signal by computing the unique XOR of the syscall number and return code. For the purposes of equal comparison, we choose to disable kernel coverage collection for all of the runtimes under evaluation and use the fallback mechanism everywhere.

3.2 Deploying Containers

Syzkaller requires that certain information be passed back from each VM to monitor for crashes and log which programs are running. We begin by implementing a dummy VM translation layer for Syzkaller that forwards commands directly onto the host and passes logs from the fuzzer back for analysis. We also introduce a small library to Syzkaller to support creating containers with arbitrary resource restrictions and runtimes. Rather than directly interact 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 of the CLI and the Docker Daemon, as well as compatible with equivalent container engines like ‘podman‘, which use the same commands. The initial Torpedo library contains support for the docker command line options given in Table 3.1. A diagram of the design is presented in Figure 3.1, suitable for comparison with Figure 2.3.

parameter	function
runtime	the container runtime
cpuset-cpus	the physical cores the container can be scheduled on
cpus	the amount of CPU utilization

Table 3.1: Docker resource restrictions supported by TORPEDO

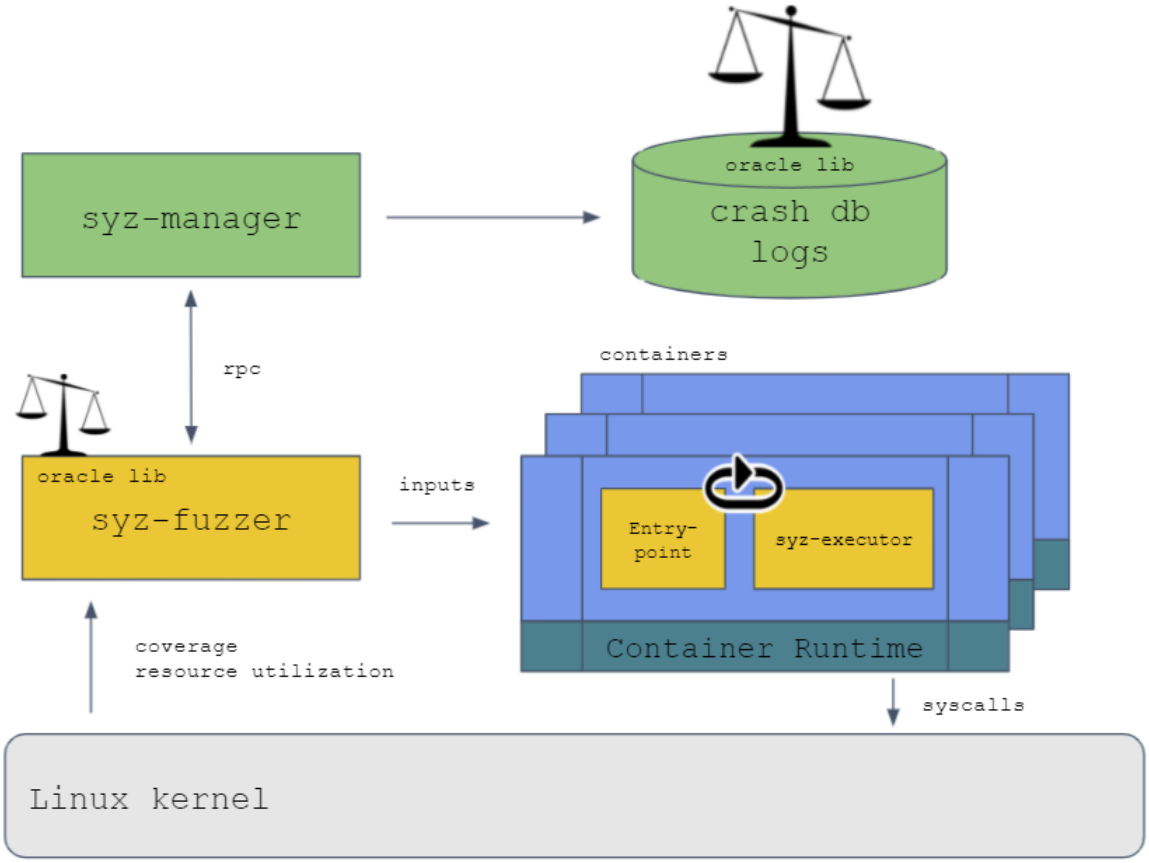

Figure 3.1: TORPEDO Architecture

3.3 Deploying Workloads to Containers

For ease of use, we package the executor binary along with a custom entrypoint binary that implements algorithm 1 into a container image. The entrypoint binary acts as a thin translation layer on the existing Syzkaller API for translating serialized programs between the fuzzer and executors. Execution requests are passed to the container via IPC pipes and results are returned using the same mechanism. We note that use of the Docker CLI to deploy containers and stream output results in a non-trivial workload being placed on the docker engine. Gao et al. observe in [21] that this results from data being flushed to the LDISC layer of the TTY subsystem through work queues managed by kernel threads, which is another example of cgroup escape via kernel deferral (Section 2.4.3). Torpedo minimizes this overhead by reducing the amount of data exchanged between the executor container and the syz-fuzzer process while measurements are being taken, although it is non-zero.

In order to capture accurate resource utilization measurements, Torpedo must 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 test have different run times as a result 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. We conclude that the only option for accurately measuring multiple fuzzing processes in parallel is to completely synchronize the program execution window, as well as extend the execution time for each program to become comparable. We choose to have the container entrypoint binary be responsible for this synchronization by implementing algorithm 1. Using this logic, we can be sure that all parallel executor containers will stop executing at or

before timestamp $\mathcal{T}$.

Algorithm 1 Configurable workload loop control. Loop an arbitrary sequence of system calls $\mathcal{P}$ until timestamp $\mathcal{T}$. Report number of executions and average execution time. All time calculations use Unix NS timestamps.

```
1: function LoopUntilTime(P , T )
2:     AvgExecutionTime ← 0
3:     TotalExecutionTime ← 0
4:     NumExecutions ← 0
5:     for ∞ do
6:         ExecutionTime ← Execute(P)
7:         NumExecutions + +
8:         TotalExecutionTime ← TotalExecutionTime + ExecutionTime
9:         AvgExecutionTime ← TotalExecutionTime/NumExecutions
10:        if CurrTimeStamp + AvgExecutionTime > T then
11:            return NumExecutions, AvgExecutionTime
```

3.4 Collecting Resource Utilization

SYZKALLER supports collection of per-syscall code coverage out of the box. In addition to this mechanism, TORPEDO is responsible for collecting resource utilization feedback for some set of resources $\mathcal{R}$.

In the native design, one thread (referred to as a 'proc') is created for each executor. These threads execute work items from the fuzzer's internal queue asynchronously and report coverage information as it becomes available. While this model is efficient, it is unsuitable for collecting an accurate "snapshot" of resource utilization as the result of a particular set of programs.

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 breaks down periods of execution into rounds, each of which lasts time T. Coordination between the observer and any number of executors is achieved using algorithm 2. This algorithm uses a two-stage latching procedure to distribute programs and prime each executor, then start the execution window to line up with some number of resource measurements. The observer has access to all feedback results, which it can use to immediately motivate changes to each program for the next round (Section 3.5). Additionally, the observer is responsible for logging this information for later analysis from another part of the Torpedo framework (Section 3.6).

The choice of the best time T is not necessarily clear. After some empirical testing, we note that too short of an interval is more easily disrupted by temporary "noise spikes" from the host (cron jobs, sudden arrival of network packets, system logging events, etc), while longer intervals produce more useful measurements but significantly reduce program throughput. We settle on values ranging a short number of whole seconds, typically between 3 and 5, as a good balance between throughput and accuracy.

In anticipation of fuzzing for adversarial CPU utilization, we create an observer 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

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

```
1: function Observer(𝒯, ℛ)
2:     RoundNum ← 0
3:     Workloads ← ∅
4:     RoundScore ← 0
5:     InitializeExecutors(ℰ)
6:     for ∞ do
7:         𝒲 ← GetPrograms(𝒲, RoundScore, ℛ)
8:         StopTime ← CurrentTime + 𝒯
9:         for ℰ ∈ Executor do
10:            ℰ.stop ← StopTime
11:            ℰ.program ← 𝒲_ℰ
12:            Signal(ℰ)
13:        WaitForAllExecutors(ℰ)        ▷ Wait for all executors to signal they are ready
14:        SignalAllExecutors(ℰ)
15:        RoundScore ← TakeMeasurement(𝒯, ℛ)        ▷ returns after 𝒯 seconds
16:        LogRoundResults(RoundScore)
17:        RoundNum++
18:
19: function Executor(𝒪)        ▷ Each executor maintains a reference to the observer
20:    program ← ∅
21:    stop ← ∅
22:    for ∞ do
23:        WaitForSignal()
24:        PrepareToExecute(program) ▷ Create a container and serialize execute request
25:        SignalObserver(𝒪)
26:        WaitForSignal()
27:        Execute(program, stop)
```

where out of band workloads are being created and tracking their efficacy. To implement this, we fork an existing Golang library [42] with a wrapper for the top(1) command [39]. We filter this output by selecting common categories of interest, such as ‘docker‘, ‘kworker‘ threads, ‘kauditd‘, ‘systemd-journal‘, and miscellaneous kernel threads.

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, for example, top would be unable to profile this usage, but we 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 out of band workloads.

3.5 Combining Feedback Mechanisms

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 it does not. A program that generates more new coverage is strictly ”better” than one that does not. The same relationship does not necessarily hold for resource utilization, however. A fuzzing input that generates more CPU utilization than its predecessor may not strictly be more adversarial; it could simply spend less time blocked. However, 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.

3.5.1 Oracle Library

We consider that the process of guiding adversarial program generation is split 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 TORPEDO's ultimate goal of identifying programs that violate one or more resource boundaries. We conceive of a library, known as an ”Oracle”, that contains the necessary logic for both of these tasks with respect to a particular resource. More formally, a resource oracle must support the following operations.

1. Score a workload. A higher score indicates the workload is more indicative of an 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 best to combine the 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. Code coverage is incorporated at the individual program level, and resource utilization at the "set of programs" level.

3.5.2 Using the Oracle: 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. However, 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.

Throughout the course of Torpedo testing, each batch of programs is guided through many rounds. Programs that do not generate new coverage are typically rejected before they spend too much time being mutated, and of the batch of programs, only the set of mutated workloads that generated the most adversarial resource usage is recorded into the corpus. To achieve this, out design splits the Syzkaller program state machine into two separate state machines. Figure 3.2 depicts the original Syzkaller state machine described in Section 2.6.3. Figure 3.3 depicts the result of dividing relevant states between the level of an individual program and a batch of programs. Notably, we implement the "confirm" phase as "shuffle",

where individual programs are shuffled between cores but the order of syscalls in each trace is not disturbed. This helps to reduce false positives from the case where where system noise is concentrated on a subset of cores and is unrelated to the program under test.

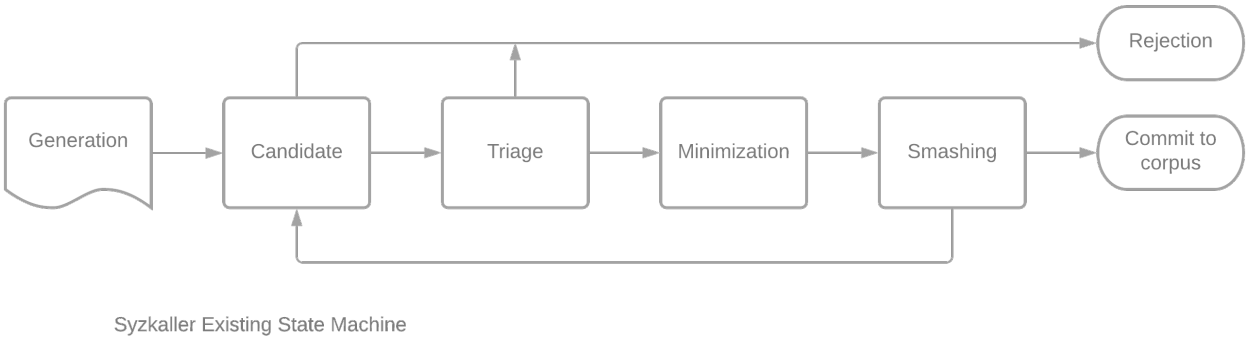

Figure 3.2: SYZKALLER Program State Machine, assessed from [58]

3.6 Identifying Adversarial Workloads

For each class of resource $\mathcal{R}$, TORPEDO must be capable of identifying what constitutes an adversarial resource utilization pattern. This is encapsulated in the second Oracle design point in Section 3.5, which requires the ability to ”flag” a workload as adversarial.

We consider that for each resource, there is a set of heuristics that can be applied to an observation to determine whether the workload is adversarial. For CPU utilization, one example would be observing a total utilization higher than the sum of all configured resource caps for a set of containers plus some system noise. Alternatively, observing a nontrivial workload placed on a core not part of the cpuset allocated to the container, or observing a greater than average load placed on a system process like ‘journald‘. Given a measurement and a set of thresholds, the Oracle can trivially process each heuristic for an observation and report whether or not one or more categories have been violated.

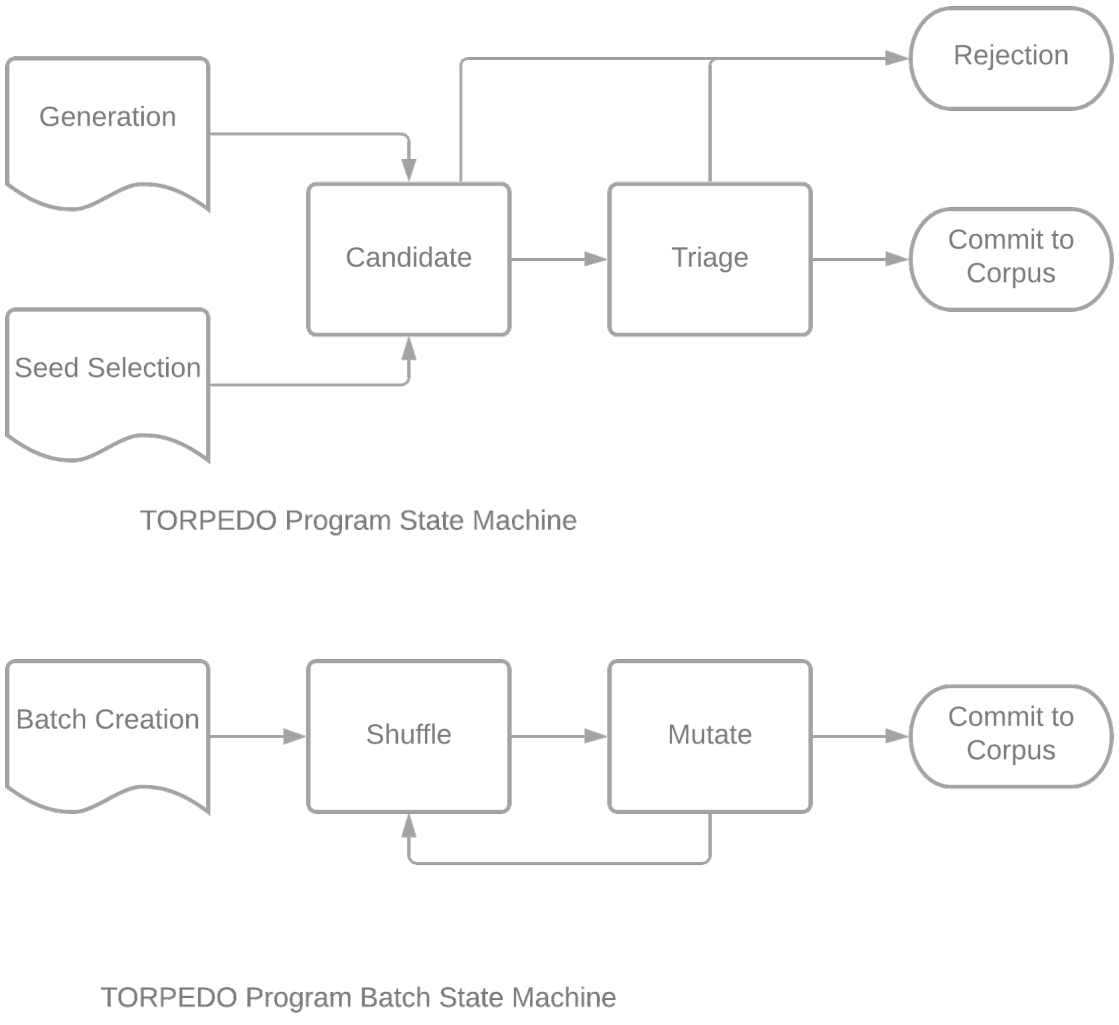

Figure 3.3: Torpedo State Machines

3.6.1 Using the Oracle: 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 asynchronously from actual 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. Once identified, a human operator can isolate the adversarial program(s) and proceed to a tool-assisted minimization workflow.

Chapter 4

Evaluation

4.1 Testing Procedure

To evaluate Torpedo, we begin by testing whether the framework is capable of identifying known vulnerabilities. We begin by distilling a handful seeds from C programs that recreate the vulnerabilities described in [21]. We use a tool packaged with Syzkaller that allows manual execution of programs in intermediate representation to confirm the system correctly identifies abnormal resource utilization trends while these programs are running, and use these observations to tune the implementation of our CPU Oracle. We extract several heuristics that should be flagged by the Oracle if violated, given in Table 4.1

4.1.1 Seed Selection

For comprehensive evaluation, we repurpose a set of high quality seeds contributed by the authors of Moonshine [47], an OS trace distillation project that improved Syzkaller's program

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

Table 4.1: Torpedo CPU Oracle Heuristics

generation capabilities. Each seed in this corpus is prepared in Syzkaller's intermediate representation and constitutes a sequence of related syscalls designed to cover a particular kernel interface. These seeds are distilled from real programs and contain call patterns that meaningfully interact with various kernel systems. By starting with realistic seed programs, we consider it more likely that our mutations will produce meaningful change in the context of the program rather than randomly access kernel modules and "skim the surface" of several unrelated code bases.

4.1.2 Container Runtimes

We choose to focus our testing on the container runtimes (Section 2.3.2 that implement the fundamental resource isolation of the container. In particular we choose to evaluate the default, "bare metal" docker runtime RUNC, as well as the virtualized runtime gVisor. We select 200 traces from the Moonshine corpus and divide them into smaller batches. We start TORPEDO with a modest number of fuzzing threads and allow it to select and fuzz programs unattended for a period of time. When each seed has been fully evaluated (it has moved through all stages given in the Program State Machine in Figure 3.3), we terminate the fuzzer and apply Oracle heuristic evaluation to the logs for each round. Interesting programs are selected and subjected to a minimization procedure given in Section 4.1.2.

We quickly observe that certain syscalls, such as 'pause', 'nanosleep', 'poll', and 'recv' send the program into the blocked state and are thoroughly uninteresting. As we observe these calls, we add them to a SYZKALLER generation denylist and filter them from seed programs.

Algorithm 3 Minimize program $\mathcal{P}$ with respect to oracle $\mathcal{O}$

```
function Minimize(P, O)
    V ← FindViolations(P, O)
    for line in P do
        P.remove(line)
        V_new ← FindViolations(P, O)
        if V ≠ V_new then
            P.replace(line)
    return P
```

4.1.3 Minimization Procedures

Interesting programs identified by the oracle are extracted and flagged for minimization. This process is guided by algorithm 3. In effect, we systematically remove calls from the program until we obtain the smallest set of calls that result in the originally observed oracle violations. In some cases, potentially unnecessary calls must be preserved to pass information to a later call that produces some interesting effect, such as a ‘read‘ call being necessary to produce a file descriptor to an error-causing ‘write‘ call. The minimization algorithm is not capable of distinguishing these cases, so minimized programs must be further evaluated to determine the root cause of the observed issue.

4.1.4 Confirmation

After confirming that a program exhibits some consistent unusual behavior when run using Torpedo, it remains to confirm that this behavior is not a quirk of the testing framework. To do so, we manually recreate the sequence of calls from the trace in C code and independently package a binary into a testing container. To avoid potential interference from optimizations or translations performed by the glibc system call wrapper functions, we use the ‘syscall(2)‘ thin wrapper to pass raw arguments directly to the kernel.

If the C-version of the trace exhibits adversarial behavior, we analyze the kernel function graph created by the program to determine what interaction is causing the initial observation. To do so, we use ftrace via the wrapper trace-cmd [17]. By searching for some of the patterns identified in [21], we can confirm the cause of the adversarial behavior and record the bug for later submission.

4.2 Experimental Setup

All experiments were conducted on a Ubuntu 20 desktop computer equipped with a Ryzen 5 3600X 12-core processor (2 threads per physical core) at 4GHz. Seeds were selected in groups ranging in size from 10 to 40. The fuzzer was allowed to run between thirty minutes and an hour at a time. Each fuzzer used 3 parallel threads and 5 second rounds. CPU Utilization was used as the Oracle score, with utilizations ranging within 2.5% of a baseline being considered equivalent to account for standard system noise. Scores had to increase by at least 1 percentage point to be considered significant. Programs were configured to cycle out after 15 rounds with no improvement.

Due to limitations in the syscall interface provided by gVisor, the requisite ioctl(2) call required to collect coverage information from the kernel is not available. To solve this, SYZKALLER includes a ”fallback” mechanism to compute coverage based on a unique combination of syscall number and error code. While this feedback is not as thorough as line coverage from inside the kernel, it suffices as a basic approximation and is suitable for proof of concept testing. We disable coverage collection on runC as well to accurately compare the fuzzing mechanism.

4.3 Results from runC

We present results from experiments conducted on the default runC container runtime in Table 4.2. The "symptoms" column indicates what conditions the syscall displays adversarial behavior. The "cause" column describes the underlying kernel interaction. The "new" column denotes whether the adversarial behavior was previously undocumented. In total we identified five categories of adversarial workload, some of which are discussed below. Examples of observation logs can be found in Appendix A.

syscall(s)	Symptoms	Cause	New?
sync, fsync	any usage	triggering IO buffer flushes	reconfirm
rt_sigreturn	any usage	core dump via SIGSEGV	reconfirm
rseq	invalid arguments	coredump via SIGSEGV	reconfirm
fallocate, ftruncate	argument exceeds max file size	coredump via SIGXFSZ	reconfirm
socket	errno {93 | 94 | 97}	repeated kernel modprobe	yes

Table 4.2: Collected Results from runC Tests

4.3.1 sync(2)

As previously described by Gao et al. in [21], the sync system call is trivially adversarial for any program running an I/O workload on the same host. The implementation triggers a flush of low-level data structures shared across multiple programs accessing the disk, which can cause those processes to become blocked while waiting for the disk to finish. The calling process is charged neither for the CPU utilization involved in flushing the data nor for the I/O bandwidth involved in transmitting the data. We observe the effects of sync and similar syscalls by noting high amounts of "I/O wait" on cores not used for fuzz testing. Table A.2 in Appendix A demonstrates an example of this adversarial pattern.

4.3.2 fallocate(2)

This example concerns a series of related function calls that manipulate the size of a file. When an process attempts to increase the size of a file beyond what is specified in its resource limits (as shown by getrlimit(2)), the kernel delivers a SIGXFSZ. The default action upon receipt of this signal is to terminate the calling process and produce a coredump. Gao et al. extensively studied the adversarial effects of the default coredump generation procedure on Linux, as described in Section 2.4.3. This side-effect of SIGXFSZ was previously unconsidered, but it logically follows that any signal which triggers a core dump would have the same effect. Namely, this includes SIGABRT/SIGIOT, SIGBUS, SIGFPE, SIGILL, SIGSEGV, SIGQUIT, SIGSYS/SIGUNUSED, SIGTRAP, SIGXCPU and SIGXFSZ by default.

4.3.3 socket(2)

This example concerns specific arguments to the socket interface inside the kernel. When creating a socket, the user supplies an argument for the family (domain), type, and protocol respectively. Depending on the configuration of the kernel, many of these options are implemented as modules which can be loaded on demand to satisfy a request. The kernel uses the aforementioned usermodehelper API (Section 2.4.3) to execute ‘modprobe‘ and bring these libraries into memory for the remainder of the boot cycle. As discovered by TORPEDO, no negative result is cached in the modprobe handling code in the event a valid socket family is requested from userspace but no corresponding module exists on disk. In this case, repeated requests for a socket will cause modprobe to be executed again and again, creating many processes that escape both CPU and CPUSET cgroup restraints. An example of this pattern is documented in Appendix A Section A.1.3. Similar errors manifest for various argument combinations which produce errors EAFNOSUPPORT, ESOCKTNOSUPPORT,

and EPROTONOSUPPORT respectively. As far as we are aware, this is the first time this behavior has been identified as adversarial in the context of containerized workloads.

4.4 Results from gVisor

We present additional results from running the same set of seeds on gVisor. In general, we observe that gVisor introduces additional overhead on syscall execution and overall utilization number are lower. An example of this can be seen by comparing Table A.4 with Table A.1 in Appendix A. This is to be expected from a virtualized runtime, which includes the usual overhead from a VM based approach.

syscall(s)	Symptoms	Cause	New?
open	container crash	invalid argument	likely
open	container crash	multithreaded collision	likely

Table 4.3: Collected Results from gVisor tests

4.4.1 open(2)

We quickly observe that the open syscall produces problems when run on gVisor. Simply perturbing the FLAGS argument to contain a specific bit pattern causes the container to crash when attempting to open a standard glibc shared object file. A recreation of this crash in C code is presented in Section A.2.2. gVisor reports on their official documentation [55] that not all flags to open(2) are supported, but quitting the container is almost certainly indicative of a bug in the underlying runtime. We observe an additional crash involving the execution of an open syscall while the container is executing several syscalls in parallel. The exact diagnosis and submission of this bug to the gVisor authors remains future work.

4.4.2 Discussion of Additional Results

Unlike runC, gVisor produced relatively uninteresting results with respect to adversarial resource utilization patterns. Notably, none of the adversarial programs identified in Section 4.3 exhibited the same behavior when run on gVisor. A survey of 300 additional seeds (to a total of 500) on gVisor failed to produce any additional results beyond the two issues already identified involving open(2). We speculate that due to the relative prevalence of open(2) in the Moonshine seeds, potentially interesting behavior may have been masked by repeated encounters with the same bug. Alternatively, as SYZKALLER was not calibrated to ignore syscalls and interfaces that are not implemented by gVisor, too much time may have been spent fuzzing shallow error parsing code rather than revealing interesting behavior. Occasionally, TORPEDO did observe potentially adversarial resource utilization on certain programs, but all attempts to reproduce this behavior failed. We unfortunately must attribute these observations to one-off quirks from the underlying system or device drivers that may be unrelated to the container system.

Chapter 5

Future Work

The work presented here shows that TORPEDO is a promising framework capable of fuzzing container runtimes, engines, and identifying adversarial workloads based on resource utilization feedback. Based on our experimentation, we have identified several opportunities for improvement going forward.

5.1 Development of Additional Oracles

While CPU Utilization is an important KPI, other Linux cgroups should be monitored as well. Having an oracle tuned to monitor memory usage and I/O bandwidth would be practical additions to the framework and allow for the discovery of additional types of adversarial workloads. Another extremely relevant metric for container systems is startup time [53], which could be monitored while workloads are running to search for correlation. How to adequately design an oracle to measure this metric while taking into account known phenomena like cold start [46] remains a task for the future.

5.2 Testing Additional Environments

runC and gVisor represent two distinctly different designs for container runtimes. However, there are other major implementations in use by industry today. To fully demonstrate the

potential of TORPEDO, we should extend testing to other runtimes and container engines. Red Hat's 'crun' implementation [13] is another example of a baremetal container runtime, and Kata [19] is another example of a virtualized runtime. Switching TORPEDO to use these runtimes for container creation would require minimal adjustments to the SYZKALLER base and would easily broaden the scope of our experiments.

Additionally, adapting TORPEDO to use a different container engine than Docker would be highly desirable. Kubernetes commanded an impressive 77% of the container orchestration market in 2019 [32] and due to the complex nature of its design, would likely benefit from a fuzzing-based approach to discover OOB workloads. Kubernetes can be configured to use practically any of the OCI runtimes that we have fuzzed via the Docker engine, although implementing a library to connect TORPEDO to Kubernetes may be challenging.

5.3 Improving Mutation Algorithm and Seeds

Although TORPEDO reasonably connects both coverage and resource utilization feedback to meaningfully guide fuzzing, the coupling of these mechanisms is not optimal. Uninteresting candidate programs are not efficiently removed from the work queue before they are fuzzed for some number of rounds, which consumes valuable time. Additionally, as SYZKALLER's mutation algorithm was designed with crash detection in mind, we consider that adjusting some of the parameters may produce some benefit. The authors note that the constants associated with the random choice of mutation are not grounded in any legitimate research, and adjusting these to achieve some degree of optimality could constitute an entirely separate project. The Moonshine seeds used for this evaluation are also likely not the most appropriate. Because they were created explicitly for the purposes of code coverage, we consider whether the inherent syscall patterns are less likely to be adversarial than traces

sourced from a subset of known adversarial programs. Finding or improving a set of seeds specifically to expose adversarial behavior also remains future work.

5.4 Testing with Proper Coverage Feedback

For the runtimes that allow it, using real kernel line coverage feedback would obviously improve the quality of the feedback used to guide program generation. Configuring the Linux kernel for coverage collection is relatively simple, although for the sake of performance, ideally this kernel would be running on bare metal and not inside a VM (as in the Syzkaller original design). It would also be interesting to collect coverage information from components of the container engine, such as the docker daemon or containerd process. On kubernetes, interesting components might be the kubelet and the API server. For virtualized runtimes like gVisor, coverage information could theoretically be collected while the containers were running, but this would be difficult to obtain from bare metal runtimes like runC, which are relatively short-lived processes.

5.5 Bug Submission

To the best of our knowledge, the adversarial behavior identified by Torpedo involving the socket interface has not previously been documented. This bug should be submitted to the Linux kernel authors as another example of the problematic nature of the usermodehelper API. Dr. Xing Gao informs us that previous attempts to raise attention to issues in the usermodehelper API have been largely ignored by the kernel community. The bugs observed in gVisor concerning the open(2) API interface need to be properly isolated and submitted to the GitHub issue tracker for the project.

Chapter 6

Summary

In this work, we presented Torpedo, a fuzzing framework suitable for identifying adversarial container workloads on Linux. Torpedo extends the existing Syzkaller OS fuzzer to support creating workloads in containers and examining system resource utilization. By combining this resource utilization feedback with existing code coverage mechanisms, Torpedo guides mutations using a genetic algorithm that promotes the development of adversarial workloads over time, then passes likely adversarial programs to a human operator for tool-assisted minimization and recreation.

We evaluated Torpedo using a corpus of realistic syscall traces from the authors of Moonshine against two commonly used container runtimes, runC and gVisor. Results from runC demonstrate that Torpedo independently reconfirmed the existence of several known vulnerabilities as well as discovered one new userspace exploit for creating an out of band CPU workload. Results from gVisor reveal at least one new bug that results in a complete container crash. These promising findings demonstrate the potential Torpedo has to improve security in the Linux kernel, container engines, and the cloud environments where these paradigms are used to conduct business.

6.0.1 Takeaways and Final Thoughts

Among security professionals, one unfortunate observation is commonly discussed: when designing any product, security is most often considered last. Over the lifetime of a product like the Linux kernel, we can see this is somewhat true just by examining the commit logs for various features. cgroups and other kernel features that enforce containerized isolation guarantees have all been added into the kernel at various times, with varying levels of compatibility with other subsystems. Until recently, the idea of complete process and resource isolation has remained within the realm of "best effort", although the extreme popularity of containers and their adoption for multi-tenant clouds has made that "best effort" a necessity. To accurately understand, design, and improve these systems, developers need to constantly be thinking about the different ways their systems will interact with existing *and* future architectures, as well as use effective testing tools. Irrespective of how the ecosystem will continue to evolve, tools like Torpedo will always be necessary to examine the fundamental guarantees of isolation frameworks and ferret out bugs, oversights, and vulnerabilities.

Designing a product like Torpedo has required learning an immense amount about the different parts of the container ecosystem, starting with the developer interface and drilling down into the implementation of specific kernel subsystems. While one of the largest selling points of a container is the ease of use the bring to developers, using one effectively requires an intimate knowledge of how the security options are translated through several processes down to the kernel. Even though Docker may reign supreme now, there's no telling what implementations will be favored as the industry continues to innovate. Hopefully the next generation of designers can gain insights through the way we tested the previous generation, and continue working toward systems that achieve complete, perfect isolation on a single host.

Bibliography

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/agache.

[2] Aymen Eon Amri. Docker, containerd |& standalone runtimes — here's what you should know. https://medium.com/faun/docker-containerd-standalone-runtimes-heres-what-you-should-know-b834ef155426, 2017. Accessed: 2021-01-25.

[3] The Kubernetes Authors. Don't panic: Kubernetes and docker. https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/, 2020. Accessed: 2021-01-30.

[4] The Kubernetes Authors. Production-grade container orchestration. https://kubernetes.io/, 2021. Accessed: 2021-01-27.

[5] Evan Baker. A comprehensive container runtime comparison. https://www.capitalone.com/tech/cloud/container-runtime/, 2020. Accessed: 2021-01-26.

[6] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 45–58, USA, 1999. USENIX Association. ISBN 1880446391.

[7] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. *Proceedings 2020 Network and Distributed System Security Symposium*, 2020. doi: 10.14722/ndss.2020.24415. URL http://dx.doi.org/10.14722/ndss.2020.24415.

[8] Emile Borel. Learned ape paradox. https://citations.webescence.com/citations/Emile-Borel/Concevons-ait-dresse-million-singes-frapper-hasard-sur-les-touches-une-machine-2819 1913.

[9] Ravid Brown. Running containers on bare metal vs. vms: Performance and benefits. https://www.stratoscale.com/blog/data-center/running-containers-on-bare-metal/, 2017. Accessed: 2021-02-04.

[10] Canonical. Apparmor. https://apparmor.net/, 2020. Accessed: 2021-01-31.

[11] corbet. Process containers. https://lwn.net/Articles/236038/, 2007. Accessed: 2021-01-21.

[12] Michael Crosby. Scope and principles. https://github.com/containerd/containerd/blob/master/SCOPE.md, 2020. Accessed: 2021-01-25.

[13] 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. Accessed: 2020-04-20.

[14] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: An empirical study at

the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 246–256, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236027. URL https://doi.org/10.1145/3236024.3236027.

[15] Jake Edge. A seccomp overview. https://lwn.net/Articles/656307/, September 2015. Accessed: 2021-04-21.

[16] Isaac Eldridge. What is container orchestration? https://blog.newrelic.com/engineering/container-orchestration-explained/, 2018. Accessed: 2021-01-27.

[17] Julia Evans. ftrace: trace your kernel functions! https://jvns.ca/blog/2017/03/19/getting-started-with-ftrace/, 2017. Accessed: 2021-04-16.

[18] Apache Software Foundation. Apache mesos. http://mesos.apache.org/, 2021. Accessed: 2021-01-30.

[19] OpenStack Foundation. Kata containers. https://katacontainers.io/collateral/kata-containers-1pager.pdf, 2017. Accessed: 2021-01-27.

[20] Xing Gao, Zhongshu Gu, M. Kayaalp, D. Pendarakis, and H. Wang. Containerleaks: Emerging security threats of information leakages in container clouds. *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 237–248, 2017.

[21] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. 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, page 1073–1086, New York, NY, USA, 2019. Association for Computing Machinery. ISBN

9781450367479. doi: 10.1145/3319535.3354227. URL https://doi.org/10.1145/3319535.3354227.

[22] Google. american fuzzy lop. https://github.com/google/AFL, 2021. Accessed: 2021-02-08.

[23] Red Hat. What is podman? https://podman.io/whatis.html, 2019. Accessed: 2021-01-24.

[24] Red Hat. What is selinux? https://www.redhat.com/en/topics/linux/what-is-selinux, 2021. Accessed: 2021-01-31.

[25] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[26] Alexander Holbreich. Docker components explained. http://alexander.holbreich.org/docker-components-explained, 2018. Accessed: 2021-01-26.

[27] Docker Inc. Developers bring their ideas to life with docker. https://www.docker.com/why-docker, 2021. Accessed: 2021-01-24.

[28] Docker Inc. Apparmor security profiles for docker. https://docs.docker.com/engine/security/apparmor/, 2021. Accessed: 2021-02-04.

[29] Docker Inc. Docker overview. https://docs.docker.com/get-started/overview/, 2021. Accessed 2021-01-24.

[30] Docker Inc. Seccomp security profiles for docker. https://docs.docker.com/engine/security/seccomp/, 2021. Accessed: 2021-04-21.

[31] Docker Inc. Isolate containers with a user namespace. https://docs.docker.com/engine/security/userns-remap/, 2021. Accessed: 2021-01-22.

[32] T4 Labs Inc. Container platform market share, market size and industry growth drivers, 2018 - 2023. https://www.t4.ai/industries/container-platform-market-share, 2019. Accessed: 2021-04-20.

[33] Alexander Kedrowitsch, Danfeng (Daphne) Yao, Gang Wang, and Kirk Cameron. A first look: Using linux containers for deceptive honeypots. In *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*, SafeConfig '17, page 15–22, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450352031. doi: 10.1145/3140368.3140371. URL https://doi.org/10.1145/3140368.3140371.

[34] Michael Kerrisk. Namespaces in operation, part 5: User namespaces. https://lwn.net/Articles/532593/, 2013. Accessed: 2021-01-22.

[35] Michael Kerrisk. *cgroups - Linux control groups*, August 2020.

[36] Michael Kerrisk. *namespaces - overview of Linux namespaces*, November 2020.

[37] Michael Kerrisk. *subuid - the subordinate uid file*, December 2020.

[38] Michael Kerrisk. *sync, syncfs - commit buffer cache to disk*, August 2020.

[39] Michael Kerrisk. *top - display Linux processes*, September 2020.

[40] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based CPU in container environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 313–328, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/nsdi18/presentation/khalid.

[41] Capsule8 Labs. An exercise in practical container escapology. https://capsule8.com/blog/practical-container-escape-exercise/, 2021. Accessed: 2021-04-21.

[42] Gyuho Lee. linux-inspect implements various linux inspecting utilities. https://github.com/gyuho/linux-inspect, 2018. Accessed: 2021-04-20.

[43] Barton P. Miller. Fuzz testing of application reliability. http://pages.cs.wisc.edu/~bart/fuzz/, 2021. Accessed: 2021-02-07.

[44] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262631857.

[45] MITRE. Cve-2017-5753. Available from MITRE, CVE-2017-5753, Februrary 2018. Accessed: 2020-02-16.

[46] Gonçalo Neves. Keeping functions warm - how to fix aws lambda cold start issues. https://www.serverless.com/blog/keep-your-lambdas-warm, 2018. Accessed: 2021-04-20.

[47] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 729–743. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor.

[48] Jackson Pauls. *Security Enhanced Linux Policy for the container processes*, December 2020.

[49] LLVM Project. libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2021. Accessed: 2021-02-14.

[50] 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 *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 199–212, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605588940. doi: 10.1145/1653662.1653687. URL https://doi.org/10.1145/1653662.1653687.

[51] Inc Software in the Public Interest. *systemd-cgtop - Show top control groups by their resource usage*, December 2020.

[52] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 275–287, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936363. doi: 10.1145/1272996.1273025. URL https://doi.org/10.1145/1272996.1273025.

[53] Dylan Stamat. The overhead of docker run. https://blog.iron.io/the-overhead-of-docker-run/, 2016. Accessed: 2021-04-20.

[54] the gVisor authors. What is gvisor? https://gvisor.dev/docs/, 2021. Accessed: 2021-01-27.

[55] the gVisor Authors. Linux/amd64. https://gvisor.dev/docs/user_guide/compatibility/linux/amd64/#open, 2021. Accessed: 2021-04-20.

[56] Steven J. Vaughan-Nichols. What is docker and why is it so darn popular? https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/, 2018. Accessed: 2021-01-24.

[57] Dmitry Vyukov. kernel: add kcov code coverage. https://lwn.net/Articles/674854/, February 2016. Accessed: 2021-02-14.

[58] Dmitry Vyukov. syzkaller - kernel fuzzer. https://github.com/google/syzkaller, 2021. Accessed: 2021-02-14.

[59] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512, 2010. doi: 10.1109/SP.2010.37.

[60] Michał Zalewski. american fuzzy lop (2.52b). https://lcamtuf.coredump.cx/afl/, 2021. Accessed: 2021-02-014.

Appendices

Appendix A

Torpedo Observer Logs

The following logs have been constructed by sampling the contents of /proc/cpu/status at two different intervals and computing the difference. The "BUSY" column is computed as the sum of all non-idle columns. We notice a persistent "SOFTIRQ" workload on the core immediately following the first non-fuzzing core; this workload is a side-effect of our framework and can be safely ignored for most analysis.

A.1 Observations from runC

A.1.1 Baseline Utilization for 3 Processes

```
program 0
mmap(&(0x7f0000000000/0x1000)=nil, 0x1000, 0x3, 0x32, 0xffffffffffffffff, 0x0)
creat(&(0x7f0000000000)='mntpoint/tmp\x00', 0x124)

program 1
r0 = inotify_init()
ioctl$FS_IOC_GETVERSION(r0, 0x80087601, &(0x7f0000000100))
alarm(0x4)
r1 = open(&(0x7f00000000a9)='/proc/sys/fs/mqueue/msg_max\x00', 0x2, 0x0)
```

```
lseek(r1, 0xfffffffffffffffb, 0x1)

lseek(r1, 0x0, 0x0)

read(r1, &(0x7f00000000e5)=""/7, 0x7)

write(r1, &(0x7f00000000ec)='47530\x00', 0x6)

ioctl$DRM_IOCTL_MODE_SETGAMMA(r1, 0xc02064a5, &(0x7f00000000c0)={0x7, 0x5,

&(0x7f0000000000)=[0x0, 0x98, 0x2, 0x0, 0x101], &(0x7f0000000040)=

[0x0, 0x8, 0xfff, 0x3, 0xfff, 0x7, 0x4, 0x9], &(0x7f0000000080)=[0x0, 0x2]})


program 2

mmap(&(0x7f0000000000/0x4000)=nil, 0x4000, 0x3, 0x20010, 0xffffffffffffffff, 0x0)

getrlimit(0x3e8, &(0x7f0000000000))
```

CORE	BUSY	TOTAL	PERCENT	USER	NICE	SYSTEM	IDLE	IO WAIT	IRQ	SOFTIRQ	STEAL	GUEST	GUEST NICE
cpu0	445	535	83.18	94	0	336	90	8	0	7	0	0	0
cpu1	443	524	84.54	85	0	349	81	9	0	0	0	0	0
cpu2	459	527	87.10	100	0	357	68	2	0	0	0	0	0
cpu3	135	634	21.29	12	0	9	499	7	0	107	0	0	0
cpu4	24	527	4.55	9	0	7	503	4	0	4	0	0	0
cpu5	23	526	4.37	12	0	6	503	4	0	1	0	0	0
cpu6	37	529	6.99	16	0	13	492	7	0	1	0	0	0
cpu7	34	528	6.44	16	0	9	494	9	0	0	0	0	0
cpu8	24	527	4.55	14	0	7	503	3	0	0	0	0	0
cpu9	34	529	6.43	16	0	12	495	6	0	0	0	0	0
cpu10	37	530	6.98	14	0	14	493	7	0	2	0	0	0
cpu11	31	529	5.86	11	0	14	498	5	0	1	0	0	0
CPU	1727	6445	26.80	399	0	1134	4718	70	0	124	0	0	0

Table A.1: Standard Utilization for 3 Fuzzing Processes under runC

A.1.2 Adversarial I/O Workload Caused by sync(2)

```
program 0

sync()


program 1
```

```
r0 = getpid()
kcmp(0x1586, r0, 0x9, 0x0, 0x0)

program 2
mmap(&(0x7f0000000000/0x1000)=nil, 0x1000, 0x3, 0x32, 0xffffffffffffffff, 0x0)
readlink(&(0x7f0000000000)='./test_eloop/test_eloop/test_eloop/test_eloop/
test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/
test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/
test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/
test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/
test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/test_eloop/
test_eloop/test_eloop/test_eloop/test_eloop\x00', &(0x7f00000001db), 0x0)}
```

CORE	BUSY	TOTAL	PERCENT	USER	NICE	SYSTEM	IDLE	IO WAIT	IRQ	SOFTIRQ	STEAL	GUEST	GUEST NICE
cpu0	220	522	42.15	35	0	151	302	32	0	2	0	0	0
cpu1	447	530	84.34	94	0	348	83	5	0	0	0	0	0
cpu2	439	531	82.67	100	0	332	92	4	0	3	0	0	0
cpu3	86	595	14.45	13	0	5	509	1	0	67	0	0	0
cpu4	20	530	3.77	13	0	5	510	1	0	1	0	0	0
cpu5	19	529	3.59	12	0	6	510	1	0	0	0	0	0
cpu6	72	529	13.61	8	0	9	457	53	0	2	0	0	0
cpu7	197	527	37.38	14	0	15	330	165	0	3	0	0	0
cpu8	17	529	3.21	12	0	3	512	2	0	0	0	0	0
cpu9	21	527	3.98	13	0	8	506	0	0	0	0	0	0
cpu10	22	529	4.16	13	0	8	507	1	0	0	0	0	0
cpu11	21	513	4.09	12	0	8	492	0	0	1	0	0	0
CPU	1587	6399	24.80	341	0	899	4812	267	0	80	0	0	0

Table A.2: Impact of Adversarial IO Behavior on Core 7

A.1.3 Adversarial OOB Workload

```
program 1
r0 = socket$netlink(0x10, 0x3, 0x9)
```

```
socketpair(0x4, 0x3, 0x7, &(0x7f0000000100))

sendto(r0, &(0x7f0000000000)="240000006004050000000000000000007

4657374696e6720617564697420737974656d", 0x24, 0x0, 0x0, 0xc)
```

CORE	BUSY	TOTAL	PERCENT	USER	NICE	SYSTEM	IDLE	IO WAIT	IRQ	SOFTIRQ	STEAL	GUEST	GUEST NICE
cpu0	52	526	9.89	10	0	13	474	2	0	27	0	0	0
cpu1	338	504	67.06	89	0	248	166	0	0	1	0	0	0
cpu2	37	503	7.36	10	0	18	466	2	0	7	0	0	0
cpu3	80	505	15.84	37	0	35	425	1	0	7	0	0	0
cpu4	76	497	15.29	32	0	40	421	3	0	1	0	0	0
cpu5	62	501	12.38	27	0	33	439	0	0	2	0	0	0
cpu6	24	503	4.77	7	0	12	479	1	0	4	0	0	0
cpu7	20	494	4.05	7	0	9	474	4	0	0	0	0	0
cpu8	27	496	5.44	9	0	11	469	6	0	1	0	0	0
cpu9	52	500	10.40	21	0	31	448	0	0	0	0	0	0
cpu10	38	493	7.71	16	0	21	455	1	0	0	0	0	0
cpu11	44	500	8.80	22	0	22	456	0	0	0	0	0	0
CPU	848	6020	14.09	286	0	493	5172	19	0	50	0	0	0

Table A.3: OOB Workload Created by Program on Core 1

A.2 Observations from gVisor

A.2.1 Baseline Utilization for 3 Processes

```
program 0

mmap(&(0x7f0000000000/0x1000)=nil, 0x1000, 0x3, 0x32, 0xffffffffffffffff, 0x0)

chmod(&(0x7f0000000000)='testdir_1\x00', 0x1ff)


program 1

setuid(0xfffe)


program 2

mmap(&(0x7f0000000000/0x1000)=nil, 0x1000, 0x3, 0x32, 0xffffffffffffffff, 0x0)
```

```
creat(&(0x7f0000000000)='getxattr01testfile\x00', 0x1a4)
setxattr(&(0x7f0000000013)='getxattr01testfile\x00',
&(0x7f0000000026)=@known='system.posix_acl_access\x00',
&(0x7f0000000033)='this is a test value\x00', 0x15, 0x1)
getxattr(&(0x7f0000000048)='getxattr01testfile\x00',
&(0x7f000000005b)=@known='system.posix_acl_access\x00', &(0x7f000000006a), 0x0)
getxattr(&(0x7f000000006a)='getxattr01testfile\x00',
&(0x7f000000007d)=@known='system.posix_acl_access\x00', &(0x7f000000008a), 0x0)
getxattr(&(0x7f000000008a)='getxattr01testfile\x00',
&(0x7f000000009d)=@known='system.posix_acl_access\x00',
&(0x7f00000000aa)=""/21, 0x15)
```

CORE	BUSY	TOTAL	PERCENT	USER	NICE	SYSTEM	IDLE	IO WAIT	IRQ	SOFTIRQ	STEAL	GUEST	GUEST NICE
cpu0	353	454	77.75	105	0	235	101	8	0	5	0	0	0
cpu1	326	449	72.61	95	0	226	123	3	0	2	0	0	0
cpu2	331	450	73.56	96	0	228	119	3	0	4	0	0	0
cpu3	72	502	14.34	8	0	5	430	2	0	57	0	0	0
cpu4	15	449	3.34	8	0	4	434	2	0	1	0	0	0
cpu5	23	445	5.17	16	0	7	422	0	0	0	0	0	0
cpu6	18	445	4.04	11	0	5	427	2	0	0	0	0	0
cpu7	20	446	4.48	14	0	2	426	4	0	0	0	0	0
cpu8	21	448	4.69	14	0	4	427	2	0	1	0	0	0
cpu9	19	448	4.24	13	0	5	429	1	0	0	0	0	0
cpu10	19	446	4.26	12	0	5	427	1	0	1	0	0	0
cpu11	20	449	4.45	11	0	5	429	2	0	2	0	0	0
CPU	1238	5432	22.79	404	0	731	4194	31	0	72	0	0	0

Table A.4: Standard Utilization

A.2.2 Crash-causing Program using open(2)

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#include <unistd.h>
#include <sys/syscall.h>

int main() {
 //original syzkaller trace
 //open(&(0x7f0000000000)='/lib/x86_64-Linux-gnu/libc.so.6\x00', 0x680002, 0x20)
  int result = syscall(SYS_open, "/lib/x86_64-Linux-gnu/libc.so.6", 0x680002, 0x20);
  printf("result was %d\n", result);
}
```