

Houdini: Benchmarking Container Security Confinement

Huzaifa Patel
School of Computer Science
Carleton University
Ottawa, Canada

David Barrera
School of Computer Science
Carleton University
Ottawa, Canada

Anil Somayaji
School of Computer Science
Carleton University
Ottawa, Canada

Abstract—While container-based workloads are now a standard part of cloud infrastructure, container security remains a challenge. Container confinement is a particularly pressing problem, as without it, a single vulnerable application can be used to compromise entire clusters of containers. Many container confinement technologies are available, but currently there is no easy way to verify whether a given configuration provides even a basic level of protection. This paper presents Houdini, a security test suite for container confinement. While it can be used to test deployable containers, Houdini is optimized for testing and comparing container confinement technologies. This paper presents the motivation, design, implementation, and initial results of running Houdini on a set of Docker container configurations. By providing a benchmark framework by which container confinement technologies can be evaluated, we believe Houdini can help foster the development of next-generation container confinement technologies.

I. INTRODUCTION

Container-based workloads have become a cornerstone of modern cloud infrastructure, revolutionizing how applications are developed, deployed, and scaled, particularly within microservices architectures and DevOps workflows. Unlike hypervisor-based virtual machines (VMs), which each require a full operating system and rely on a hypervisor for hardware virtualization, containers share the host system’s kernel, reducing their overhead. This efficiency makes them exceptionally well-suited for continuous integration and rapid scaling, allowing organizations to optimize resource utilization and reduce costs while ensuring quick, consistent deployments. Furthermore, containers enable seamless portability by packaging all userspace dependencies into a unified image, thereby avoiding the common problem of environment-specific discrepancies. Yet, despite containers’ numerous benefits, they also introduce new security challenges, especially where container confinement is concerned.

Linux kernel features such as cgroups and namespaces provide resource, process, and network isolation for containers. However, this separation is not foolproof. Kernel vulnerabilities, such as buffer overflows or privilege escalation flaws can allow a process to circumvent the very mechanisms designed to confine it. Misconfigurations can similarly open holes in container security, for instance by granting

unnecessary capabilities, mounting sensitive host directories, or running containers in privileged mode. Container escapes exploit these weaknesses to break out of the container’s resource and namespace isolation. Once an attacker gains elevated privileges or direct access to the host system, they can disrupt other containers, exfiltrate data, or compromise the entire infrastructure.

Technologies like SELinux, AppArmor, and seccomp provide additional layers of host-based security to mitigate these issues but are notoriously prone to misconfiguration. Best practices guides for container security [1] can help but they offer no guarantee that a given configuration will perform as intended by the end user. As container adoption has skyrocketed across industries, particularly in multi-tenant cloud environments, these security challenges have become more pronounced.

To improve container security, the cloud industry has developed technologies like Kata Containers¹, which isolate containers further by running them within lightweight VMs. While this approach can be effective, it also adds another layer of complexity which is subject to its own set of vulnerabilities and implementation errors. Simply adding additional layers of security does not, in itself, mean that security goals have been achieved.

Test suites play a crucial role in verifying that systems meet specified requirements and function correctly under defined conditions. They are widely used across computing disciplines, from performance benchmarking to security assessment. For instance, the Standard Performance Evaluation Corporation (SPEC) [2] is best known for developing industry-standard benchmarks that evaluate CPU performance and power efficiency, while the Transaction Processing Performance Council (TPC) [3] establishes benchmarking standards for database performance. Specialized security-focused test suites play a vital role in detecting vulnerabilities in deployed systems. Tools such as network scanners aid in identifying misconfigurations, outdated software versions, and unintended service exposures, helping organizations proactively mitigate security risks. However, a significant gap remains in the context of containers. There are currently no standardized or widely adopted test suites designed specifically to evaluate container

This paper appears in proceedings of the Privacy Security and Trust (PST) 2025 conference. This is the author’s copy for personal use.

¹<https://katacontainers.io>

confinement, functionality, or isolation effectiveness. This absence leaves organizations to rely on ad hoc methods, which are often inadequate for uncovering subtle misconfigurations or ensuring robust confinement. Given the growing reliance on containerized environments in modern infrastructure, the development of comprehensive test suites tailored for containers is imperative.

In this paper, we describe *Houdini*, the first test suite for verifying container confinement. Given a Docker container configuration (including the kernel, Docker, and runc versions), *Houdini* will instantiate the container in a standalone QEMU-based VM and perform multiple tests (or *tricks*, as we call them) to evaluate the container’s security posture. These tests explore vulnerabilities such as improper resource allocation, inadequate isolation, and insecure filesystem setups. *Houdini* is written in Python and provides an extension language for writing tricks, allowing it to adapt as new container escape vulnerabilities arise.

This paper makes the following contributions:

- Defines container confinement and its associated challenges.
- Introduces *Houdini*, a modular framework for testing container security.
- Demonstrates *Houdini*’s ability to detect security issues, misconfigurations, and inconsistencies in container confinement.

The rest of this paper proceeds as follows. Section II defines container confinement and associated challenges. Section III further motivates the need for a container security test suite. We describe *Houdini*’s design and implementation in Sections IV and V. Section VI presents three case studies demonstrating how *Houdini* can detect security issues, misconfigurations, and other inconsistencies. We review related work in Section VIII and conclude in Section IX.

II. THE CONTAINER CONFINEMENT PROBLEM

Container confinement can be defined as the set of mechanisms and strategies used to isolate a containerized application from the host system and other containers. Effective confinement is critical for maintaining system security and stability, as it ensures that containers operate within defined boundaries by restricting their access to system resources including processes, networks, filesystems, devices, and kernel interfaces.

A challenge with container confinement is that the level of confinement of a container is not fixed; it can vary based on the specific requirements of the user. In other words, the degree to which a container is isolated from the host system or other containers is customizable to meet the user’s needs. At one end of the spectrum, one can lock a container down so tightly that it is almost entirely isolated from the host system, which minimizes potential attack surfaces but can also render the container incapable of performing key tasks. On the other hand, giving a container too many privileges increases the impact of any security compromise. The optimal configuration

is one that strikes a balance between maximum confinement (i.e., least privilege) and sufficient functionality.

A standardized, formal approach involving rigorously evaluating and analyzing the security of the container and its environment might seem ideal; however, it is impractical because container environments are highly complex, with interactions between kernels, runtimes, orchestration tools, and external dependencies that are difficult to model comprehensively. Moreover, many vulnerabilities stem from subtle details that a formal model might overlook. Instead, we believe an empirical approach offers a practical alternative. Although it will likely never be exhaustive, the goal is not comprehensive coverage but rather establishing a baseline set of tests that demonstrate effective container confinement.

III. MOTIVATION

Houdini is designed to provide a systematic and reproducible approach to testing container confinement mechanisms. One of its primary goals is to evaluate whether container security features, such as namespaces, cgroups, seccomp, and Linux capabilities are effectively enforcing isolation. Many container security tools rely on theoretical guarantees or static analysis [4], but *Houdini* focuses on empirical validation by executing controlled security tests that verify the correct enforcement of security mechanisms. By doing so, *Houdini* ensures that container security mechanisms are correctly configured.

Houdini is not only for testing container isolation but also as a means to optimize container configurations. By systematically simulating various tasks within different containers, *Houdini* can help determine the minimal set of privileges required for a container to perform its intended function. For example, administrators can start with a container configured with broad privileges and then incrementally reduce permissions, such as Linux capabilities, seccomp filters, or resource limits, while monitoring task success. *Houdini*’s testing framework can help identify the point at which functionality is maintained while unnecessary privileges are removed, thereby pinpointing the least privilege configuration that still allows the container to operate effectively. Minimizing the privileges granted to a container reduces its attack surface, making it more difficult for an attacker to exploit vulnerabilities.

Although runtime monitoring and anomaly detection, such as with eBPF-based tracing [5] can provide immediate insights into container behavior, our goal isn’t to implement an exhaustive, continuous surveillance system. Instead, we aim to focus on auditing and validating the underlying isolation mechanisms to ensure they are correctly configured and consistently enforced.

A. Example Applications of *Houdini*

A system like *Houdini* can be used in the following ways: **Identify gaps between intended confinement policies and actual enforcement:** There is often a discrepancy between the security policies that administrators intend to enforce and what is actually upheld at runtime. These discrepancies

generally fall into two categories: implicit policy problems and conflicting policy configurations.

An implicit policy problem occurs when an administrator doesn't explicitly set a security rule, and the system's default behavior is used instead. These defaults might lead to behavior the administrator did not expect. For example, Docker containers are, by default, connected to a shared internal bridge network. While this topology typically prevents internet access, it still allows containers to communicate with each other. An administrator might see that outbound traffic from a container is blocked and mistakenly believe the container is fully isolated from other networks. However, internal communication between containers will still be allowed unless it's explicitly disabled. Thus, this default behavior can create a false sense of security and introduce unexpected risks.

A conflicting policy configuration, on the other hand, occurs when a security policy is correctly specified but is later invalidated by other conflicting configurations. For example, an administrator might launch a container with `--network=none`, expecting it to be fully isolated from all network communication. However, if the container is also inadvertently given elevated privileges such as `--cap-add=NET_ADMIN` or `--privileged`, it can reconfigure its own network interfaces and regain connectivity. With the `NET_ADMIN` capability, processes inside the container gain access to low-level networking operations, such as creating virtual interfaces, assigning IP addresses, or modifying routing tables. A process could use commands like `ip link set eth0 up` and `dhclient eth0` to bring up a previously disabled interface and request an IP address from the host's network. In the case of a `--privileged` container, even more capabilities are granted, allowing near-complete access to the host's kernel features, including the creation of new network namespaces or interfaces, effectively bypassing the isolation intended by `--network=none`.

Houdini can help identify these gaps by systematically testing the behavior of the container against the intended security policies, as demonstrated in case study 2 in section VI.

Challenge misleading security assumptions with empirical testing: Container security is often discussed in terms of theoretical guarantees, with many security best practices based on assumptions rather than direct testing. For instance, documentation may state that a container running in unprivileged mode cannot escalate to root, but Houdini actively tests whether that is actually the case under different container configurations. By running real-world security validation instead of relying solely on security claims, Houdini helps challenge misleading assumptions and exposes where security mechanisms fail in practice. This is particularly important as container escapes and privilege escalation exploits continue to emerge, demonstrating that security controls do not always work as expected. By grounding security claims in measurable results rather than theory, Houdini helps researchers and practitioners develop more robust container security strategies.

Detect security drift over time: One of the challenges in maintaining secure containerized environments is configura-

tion drift. Security settings applied during container initialization may be inadvertently changed through image updates, runtime automation, or infrastructure changes. Houdini can be set up to periodically re-evaluate security policies, making it possible to detect when the system diverges from its originally intended security posture. This allows for the timely remediation of regressions that could otherwise go unnoticed. Configuration drift does not only arise from explicit policy changes, but also from underlying updates to system components such as the Linux kernel, Docker, or runc. Even when container configurations and security policies remain unchanged, an upgrade to the kernel or container runtime can subtly alter the enforcement of isolation primitives, security defaults, or available capabilities. For example, a new kernel version may modify the behavior of namespaces, cgroups, or seccomp filters; a Docker update may introduce new features or alter default permissions.

B. Design Goals

In order to support the use cases mentioned above, we designed Houdini with the following goals:

- 1) *Reproducible Results.* A key goal of Houdini is to ensure that tests yield consistent and repeatable results. By controlling system variables and maintaining a structured testing environment, Houdini allows researchers and practitioners to compare results across different Docker configurations and security policies.
- 2) *Separation from the Host.* Houdini is designed to run tests inside a controlled containerized environment within a VM, ensuring that even if a trick triggers a security vulnerability, it does not compromise the host system running the tests. This design choice enhances the safety of security evaluations, preventing unintended side effects on the underlying infrastructure.
- 3) *Test Case Expressiveness.* Houdini tricks should be maximally expressive, such that some combination of steps can be used to achieve and test any desired result. It should be possible to define a new Houdini trick and modify existing tricks without modifying the Houdini binary. Moreover, it should be clear from reading a defined trick precisely what steps are involved, the consequences of each step passing or failing, and the overall nature of the exploit being tested.
- 4) *Focus on Observing Failures, Not Preventing Them.* Houdini is built to identify security weaknesses rather than defend against attacks.

IV. DESIGN

Houdini is built using Python 3 and leverages a QEMU VM to provide a controlled testing environment for container security validation. We use a QEMU-based VM for two reasons: accurate execution environment and cross-architecture support. First, QEMU supports hardware-assisted virtualization, providing a near-native execution environment that closely mirrors real-world behavior. Second, when hardware support is unavailable, QEMU enables full system emulation,

allowing tests to be conducted on architectures different from the host CPU.

Instead of running containers directly on the host system, Houdini starts a dedicated VM, ensuring that any security tests remain isolated and do not affect the host system. Dedicated VMs also help ensure test reproducibility. Within the Houdini configuration, specific versions of Docker, the Linux kernel, and runc can be requested and installed into the VM². The VM is created with the help of Buildroot [6], which enables the construction of minimal, reproducible Linux environments. This granularity in configuration enables our goal of easily testing how different kernel and container runtime versions impact policy enforcement. It also enables regression testing across versions to detect when security behavior changes unexpectedly.

At the core of Houdini’s architecture (see Figure 1), a Python Flask-based server runs on the host operating system. This server acts as a bridge between the host and the VM, facilitating communication between them over a lightweight HTTP API. The Flask server is responsible for sending commands and configuration data to the VM and receiving trick execution results. When a trick is initiated, the Houdini client on the host instructs the VM to start a new trick. The VM then launches a container with the specified security configurations and executes the trick inside it.

When a trick is initiated, the Flask server exposes an endpoint that the Houdini client on the host uses to send a request containing the container configuration and the test logic to execute. The server then transmits this data into the VM. Inside the VM, a listener service receives this request, parses the configuration, and launches a container with the specified settings. The trick code is then executed within the container. Once execution is complete, the listener inside the VM collects the results, and sends them back to the Flask server. The Flask server then responds to the original request with the results, allowing the host to log the outcome.

If Houdini were compromised during testing, it would not necessarily pose a threat to the host system due to the design of the testing environment. Houdini is intentionally designed to operate within a controlled, containerized environment inside a VM. This means that even if a trick were to exploit a vulnerability within the containerized environment, the impact would be contained within the VM, isolated from the host system and other containers. While the VM’s primary purpose here is to provide a consistent testing environment, it also serves as an additional layer of security.

V. IMPLEMENTATION

Tricks are divided into three separate files: a configuration file, a trick file, and a Dockerfile. Each Houdini trick begins with a configuration file (see Listing 1), which contains the Docker container’s settings and environment configuration.

²While Houdini currently only works with Docker containers, we expect that other container runtimes would be straightforward to support given that most have some degree of Docker compatibility.

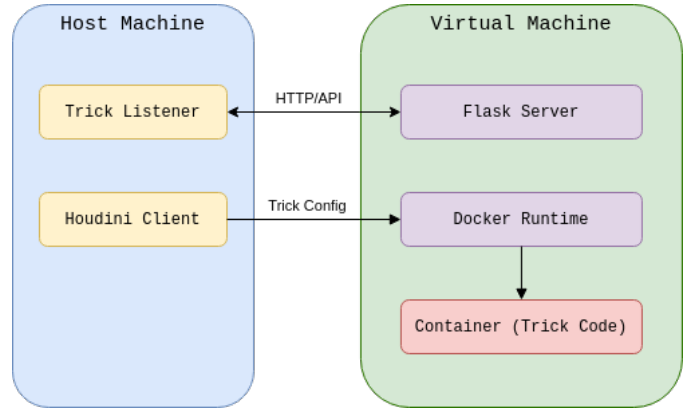


Fig. 1: Architecture of Houdini

The configuration file defines various Docker-specific parameters that are directly fed to the Docker API to start a container with the desired settings. Once the configuration is parsed, Houdini communicates these settings to the Docker API, using Docker’s run command to initiate the container with these parameters. Control over the version and configuration ensures that the container is launched identically at every run which makes it easier to test and replicate environments.

Listing 1: Configuration file for a default Docker out-of-the-box security settings applied when a container is launched.

```

name: out-of-the-box docker configuration
dockerfile:
  - path: Dockerfile
dependencies:
  - server: False
docker_config:
  - network_mode: bridge
  - read_only: False
  - security_opt: []
  - pid_mode: null
  - cpu_shares: 1024
  - volumes : []
  - mem_limit: null
  - cpuset_cpus: null
  - cpu_quota: null
  - cpu_period: null
  - cap_add: []
  - cap_drop: []
  - privileged: False
  - user: root
  - pids_limit: null
  - ipc_mode: private
  - userns_mode: null
trick:
  - path: /houdini/tricks/OutOfTheBoxExample
  
```

The second component is the trick implementation, which can be written in any programming language and is executed inside the container. This file is responsible for carrying out the actual trick scenario. After executing its logic, the trick implementation reports its output status so that Houdini can determine whether the scenario succeeded or failed. This can be done as simply as outputting a human-readable message, such as “Trick passed” or “Trick failed”. This message is then

captured by the system and used to determine the result of the test. The definition of what constitutes a pass or fail is entirely controlled by the logic within the trick implementation itself. This allows the test writer to tailor the evaluation criteria based on the context of the trick.

Listing 2: Example trick implementation

```
import os

def check_shadow_access():
    path = "/etc/shadow"
    if os.path.exists(path):
        try:
            with open(path, 'r') as f:
                print("Able to read
                ↪ /etc/shadow:")
                print("Trick passed")
        except PermissionError:
            print("Permission denied to read
            ↪ /etc/shadow.")
            print("Trick failed")
        except Exception as e:
            print(f"Unexpected error: {e}")
            print("Trick failed")
    else:
        print("/etc/shadow does not exist.")
        print("Trick failed")

check_shadow_access()
```

Finally, the Dockerfile is used to configure and set up the container’s environment. It specifies how the container should be built, including copying necessary files into the container and installing required dependencies. The Dockerfile also configures the container with the settings defined in the configuration file (see Listing 1), ensuring the container is set up correctly to run the trick.

This modular design, comprising of the configuration file, a trick implementation file, and Dockerfile provides flexibility to define and test various container escapes and vulnerabilities. It also allows users to easily customize or extend existing tricks by modifying any of the three components to suit their needs.

To illustrate how Houdini operates, a simple example trick is provided in Listing 2. This introductory trick attempts to read the `/etc/shadow` file inside the container. While not an actual exploit, this scenario serves as a baseline check for container isolation: under normal configurations, containers should not have permission to access this file. The purpose of this example is to demonstrate the structure of a trick using Houdini’s modular design, without relying on any specific vulnerability. Actual tricks and exploits are discussed in detail in the Evaluation section.

The Dockerfile defines the container’s build environment, installing any necessary packages and copying the trick script into the container. In this example, the container is based on a minimal Python image, and the trick script is executed upon container startup (see Listing 4). The trick implementation itself is a Python script (see Listing 2) that attempts to open `/etc/shadow`. If the file is accessible and readable, it prints a partial contents and reports “Trick passed”. Otherwise,

it reports “Trick failed”. This simple logic is sufficient to demonstrate Houdini’s end-to-end flow from configuration to result interpretation.

VI. EVALUATION

The case studies were done in the following environment:

TABLE I: Testing Environment

Component	Specification
Host Machine	20 Core CPU, 16GB RAM, Ubuntu 24.04
Virtual Machine	QEMU, 4 Core CPU, 12GB RAM, minimal Python image
Kernel Version	6.7

A. Case Study 1: Evaluating the accuracy of CVE-2024-21626

CVE-2024-21626 [7] is a container escape vulnerability in runc, the low-level container runtime used by Docker and other container engines. It allows a malicious container to escape its isolation and execute arbitrary commands on the host system.

The root cause of this vulnerability lies in how runc handles cgroups for containers, which are used to manage system resources like CPU and memory. During container creation, runc opens a file descriptor to the host’s `/sys/fs/cgroup` directory to configure resource limits. This descriptor remains open throughout the container’s lifecycle and is exposed inside the container via `/proc/self/fd/`, which lists all open file descriptors of the running process. The vulnerability arises when an attacker sets the working directory in the Dockerfile to point to one of these exposed descriptors, such as `/proc/self/fd/N`, where N refers to the specific file descriptor inherited from runc that points to the host’s cgroup hierarchy. This causes the container’s current working directory to resolve to a privileged host path. As a result, all subsequent relative file operations in the container are interpreted from the host’s cgroup filesystem rather than the container’s own root, allowing unintended access or modification of host resources.

This behavior is enabled by Linux’s treatment of `/proc/self/fd/N` as dynamic symlinks to open file descriptors. When dereferenced, these symlinks resolve to the original host-side path, regardless of container filesystem restrictions. Because runc and Docker do not restrict access to inherited file descriptors or sanitize the working directory path during initialization, attackers can exploit this oversight to escape isolation. By simply changing the container’s working directory to a leaked file descriptor, a malicious container can directly interact with host-managed control groups. To test the exploitability of CVE-2024-21626, we designed a Houdini trick that executes the following steps on Docker and runc versions listed in Table II:

- 1) Set the working directory to `/proc/self/fd/<fd>`, pointing to a controlled file descriptor.
- 2) Attempt to break out of the container by leveraging the misconfigured working directory.
- 3) Validate whether commands can be executed on the host system.

The CVE for this vulnerability indicates that the affected runc versions are $\geq 1.0.0\text{-rc93}$, $\leq 1.1.11$. This suggests the vulnerability was introduced in runc version rc93 and fixed in version 1.1.12. However, our tests conducted with Houdini revealed a significant discrepancy with this assertion. Our testing showed that earlier runc versions, such as 1.0.0-rc1, 1.0.0-rc2, 1.0.0-rc3, and 1.0.0-rc4 also exhibited the same vulnerability. This discovery expands the scope of the CVE by revealing that the issue existed well before version rc93.

This discrepancy highlights a recurring issue in vulnerability disclosure processes: CVEs often rely on assumptions from static code analysis or limited testing. Without rigorous empirical validation across historical versions, critical edge cases can be missed. In the case of CVE-2024-21626, this oversight could have allowed vulnerable deployments relying on older runc versions to remain unpatched under the false belief that they were unaffected.

TABLE II: runc Versions Against CVE-2024-21626

Docker Version	Runc Version	Leaked FD	Exploit Success
-	runc 0.0.9	-	✗
1.12.0	runc 1.0.0-rc1	/proc/self/fd/4	✓
17.03.0-ce	runc 1.0.0-rc2	/proc/self/fd/4	✓
17.06.1-ce	runc 1.0.0-rc3	/proc/self/fd/5	✓
17.09.0-ce	runc 1.0.0-rc4	/proc/self/fd/5	✓
-	runc 1.0.0-rc8	-	✗
-	runc 1.0.0-rc10	-	✗
-	runc 1.0.0-rc91	-	✗
20.10.17	runc 1.1.2	/proc/self/fd/9	✓
23.0.1	runc 1.1.4	/proc/self/fd/8	✓
23.0.5	runc 1.1.5	/proc/self/fd/8	✓
24.0.2	runc 1.1.7	/proc/self/fd/9	✓
24.0.5	runc 1.1.8	/proc/self/fd/9	✓
24.0.7	runc 1.1.9	/proc/self/fd/9	✓
24.0.0	runc 1.1.11	/proc/self/fd/9	✓
-	runc 1.1.12	-	✗
-	runc 1.1.13	-	✗
-	runc 1.1.14	-	✗

These findings highlight how Houdini can challenge misleading security assumptions and identify gaps between intended and actual enforcement.

B. Case Study 2 - Evaluating Host Process Manipulation

In this evaluation, we examine whether a Docker container running as a non-privileged user with UID 1000 can interact with and manipulate host processes that are also owned by UID 1000. Under a default Docker configuration, the PID namespace provides process isolation by restricting a container's visibility and access to processes outside its own namespace. However, when the container is launched with the `--pid=host` flag, it shares the host's PID namespace, allowing it to enumerate and interact with host processes. This configuration effectively bypasses one of the core isolation mechanisms provided by containers. To assess the security implications of this namespace sharing, we investigate whether process-level interactions, such as sending signals or attaching debuggers are permitted between the container and

Listing 3: Trick Implementation for Case Study I

```
#!/bin/bash
relative_path='../../../../../../'
target_file='houdini/tricks/HostMount/61240abcc5412ab'

if cd "$relative_path" 2>/dev/null; then
    echo "Successfully changed directory to: $(pwd)"
    if [ -f "$target_file" ]; then
        echo "File found: $target_file"
    else
        echo "File not found: $target_file"
    fi
else
    err_code=$?
    case $err_code in
        1)
            echo "Error: Permission denied"
            ;;
        2)
            echo "Error: No such file or directory"
            ;;
        *)
            echo "Unexpected error (exit code
            ↪ $err_code)"
            ;;
    esac
fi
echo "Docker version:"
docker --version || echo "Docker not found"

echo "runc version:"
runc --version || echo "runc not found"
```

Listing 4: Dockerfile for Case Study I

```
FROM ubuntu:16.04
WORKDIR /proc/self/fd/4
COPY host_mount.sh /host_mount.sh
RUN chmod +x /host_mount.sh
CMD ["/host_mount.sh"]
```

host processes when UID ownership is aligned. The testing environment for this case study is in Table III.

TABLE III: Case Study 2 Testing Environment

Component	Specification
Docker Version	27.0.3
Container Runtime	runc 1.1.13

Case study 2's results are summarized in Table IV, where all pairs of flags were tested using Houdini, and where the cells show what functionality was possible when launching containers with the pair of flags intersecting the rows and columns of Table IV.

Our objective here is to investigate how various security flags interact with each other, and to determine the minimum set of privileges required for a container to manipulate host processes. Specifically, we looked at the following flags:

- `--pid=host`
- `CAP_KILL`
- `CAP_SYS_PTRACE`
- `--privileged`
- `apparmor=docker-default`

The five selected flags were chosen to isolate the minimum privileges required for a container to interact with host processes. The `--pid=host` flag allows the container to see host processes, while `CAP_KILL` and `CAP_SYS_PTRACE` enable it to send signals and attach debuggers, respectively. The `--privileged` flag serves as a reference point for unrestricted access, and AppArmor's `docker-default` demonstrates how mandatory access controls can block these interactions.

Other flags, such as those affecting network, IPC, or mount namespaces, were not included because they do not impact process-level interactions. Broad capabilities like `CAP_SYS_ADMIN` were also excluded to avoid unnecessary privilege.

The evaluation began by assessing whether a container launched with `--pid=host` could list host processes using `ps aux`. As expected, the container displayed all processes running on the host, confirming that sharing the host PID namespace fully exposes the host's process table to the container.

Following this, we tested whether a container process could send a `SIGKILL` signal to a host process owned by UID 1000. The container was launched with `--pid=host`, allowing it to see and interact with host processes. Despite not being granted `CAP_KILL`, the signal was successfully delivered. This is because by default, both processes share the same UID and PID namespace, and under standard Linux permission checks, a process can send signals to another process it owns by UID without requiring special capabilities. However, if the target host process is owned by a different UID (another non-root user), `CAP_KILL` is required for the signal to be delivered. That capability must also be effective in the user namespace of the target process. Similarly, signaling host processes owned by UID 0 remains restricted even with `CAP_KILL`, unless the container process itself is also running as root.

Combining `--pid=host` with `CAP_SYS_PTRACE` enables nearly the same level of interaction with host processes as granting `--privileged` mode. Specifically, it allows a containerized process to list host processes, send signals, and attach a debugger to them, all key capabilities typically associated with privileged containers. However, unlike `--privileged`, this more targeted configuration avoids giving the container unnecessary capabilities such as access to all of the host's devices. These results demonstrate that it is unnecessary to over-privilege containers when finer-grained controls can achieve the same functional outcomes with a significantly better security posture.

When the container is confined by AppArmor's `docker-default` profile, its ability to interact with host processes becomes significantly restricted. Specifically, `docker-default` explicitly denies system calls such as `signal` and `ptrace`, which are required for sending signals and attaching debuggers to other processes. As a result, even if the container shares the PID namespace with the host and possesses the necessary capabilities, these operations will fail.

However, including the `--privileged` flag when starting the container overrides AppArmor confinement by disabling the `docker-default` profile entirely. This lifts all AppArmor-imposed restrictions, allowing the container full access to the previously restricted system calls. As a result, operations that were previously blocked, such as sending signals or attaching debuggers to host processes are now permitted, even if AppArmor would normally deny them.

It is also important to note that granting capabilities such as `CAP_KILL` and `CAP_SYS_PTRACE` alone is not sufficient for interacting with host processes unless the container shares the host's PID namespace. Without the `--pid=host` flag, the container operates in its own isolated PID namespace, meaning it cannot see or reference host PIDs at all. As a result, any attempt to send signals or attach to host processes will fail.

We note that the explanation above was derived after substantial amount of testing and analysis. The Docker documentation alone does not clarify this behavior. Instead, users are expected to understand both the Docker tooling and the underlying Linux kernel security model to correctly select these flags.

This case study highlights a core strength of Houdini, which is its ability to systematically explore and identify the least privilege for a given operation. By allowing users to declaratively modify container configurations, Houdini makes it easy to test a wide range of privilege combinations in a repeatable and automated way. Instead of relying on trial and error or defaulting to the overly permissive `--privileged` mode, users can incrementally grant permissions and observe the effect of each change. Houdini thus enables security practitioners, developers, and researchers to precisely determine the minimum set of permissions required for a given trick to succeed. As a result, Houdini not only serves as a testing framework for container escapes and misconfigurations, but also as a practical tool for privilege minimization and hardening container deployments.

C. Case Study 3 - Evaluating Docker Privileged Port Forwarding Restrictions

In this case study, we evaluate Docker's network isolation mechanisms to determine how well they enforce privileged port binding restrictions. Starting from Docker version 20.03, unprivileged ports start at 0 instead of 1024, effectively removing the need for root to bind to privileged ports. Table V presents the results of this change.

This case study highlights how discrepancies between Docker versions in handling privileged port binding can create gaps between administrators' intended confinement policies and actual enforcement. A change in Docker's handling of binding to privileged ports, such as the removal of the `NET_BIND_SERVICE` requirement starting from Docker 20.03, may allow containers to bypass restrictions even when policies are set to prevent it.

For example, administrators might configure a container to be restricted from binding to privileged ports, but if Docker is running a version where the `NET_BIND_SERVICE` capability

TABLE IV: Ability for containers to interact with host processes

Security Configuration	--pid=host	CAP_KILL	CAP_SYS_PTRACE	--privileged	apparmor=docker-default	--pid=host + apparmor=docker-default
--pid=host	◇ △	◇ △	◇ △ *	◇ △ *	◇	◇
CAP_KILL	◇ △					◇
CAP_SYS_PTRACE	◇ △ *					◇
--privileged	◇ △ *					◇ △ *
apparmor=docker-default	◇					◇
--pid=host + apparmor=docker-default	◇	◇	◇	◇ △ *	◇	◇

The following tests were run as a non-root user with UID=1000 and were run against the same non-root process with UID=1000.

◇ List Host Process

△ Send Signal to Host Process

* Attach Debugger to Host Process

TABLE V: Ability for a container to bind to privileged ports

Docker Version	NET_BIND_SERVICE + root	NET_BIND_SERVICE + non-root user	root	non-root user	Default Container
19.03.9	✓	✗	✗	✗	✗
19.03.0	✓	✗	✗	✗	✗
20.10.0	✓	✗	✗	✗	✗
20.10.3	✓	✗	✗	✗	✗
23.0.0	✓	✓	✓	✗	✗
25.0.0	✓	✓	✓	✗	✗
27.0.1	✓	✓	✓	✗	✗

Note: A non-root user implies a user account that does not have administrative privileges (not root-less Docker).

is no longer required for root containers (like in Docker 20.03 and beyond), the container could still bind to these ports despite the policy specifying otherwise. This creates a mismatch between the intended policy (restricting privileged port binding) and the actual enforcement (where containers can still bind to those ports in certain Docker versions).

Once again, Houdini helps identify these gaps by systematically testing container configurations across different Docker versions. By running different Docker versions and security policies, Houdini verifies whether containers are correctly enforcing the intended restrictions on privileged port binding. For instance, Houdini could test whether containers configured to prevent binding to privileged ports are still able to bind to those ports in Docker versions 20.03 or later, where the security model has changed. If a container running with Docker 20.03 or newer can bind to privileged ports despite the configuration specifying otherwise, Houdini will detect this discrepancy and highlight the gap between the intended policy and actual enforcement.

This case study also demonstrates a clear example of configuration drift arising not from explicit changes made by administrators, but from underlying changes in the Docker runtime itself. As Docker evolves, fundamental shifts in its privilege model, such as the redefinition of which ports are considered privileged and the removal of the NET_BIND_SERVICE requirement for certain operations can silently undermine previously established security policies. Even if an organization maintains the same container configuration and security policies, upgrading Docker to a newer version can inadvertently expand container capabilities, allowing behaviors that were once restricted. This version-induced drift can lead to a false sense of security, where administrators believe containers are still confined by earlier policy expectations, when in fact those policies are no longer fully enforced.

VII. DISCUSSION

The evaluation section demonstrated Houdini’s ability to systematically uncover gaps between intended and actual security guarantees in containerized environments. Through a series of case studies, we showed how Houdini can reveal vulnerabilities that persist across multiple runtime versions (as with CVE-2024-21626), analyze the impact of privilege and namespace flags on host interaction, and detect configuration drift resulting from runtime or policy changes. Collectively, these results highlight the value of empirical, reproducible testing for strengthening container confinement.

It is important to emphasize that Houdini’s applications extend beyond the specific scenarios examined in this paper. The modular, extensible design enables users to create and share new tricks for testing a wide range of security properties, misconfigurations, or policy violations. As a general-purpose framework, Houdini is poised to support the broader research and practitioner community in addressing evolving security challenges in containerized systems, including those not explicitly covered here.

Houdini’s design not only addresses critical security challenges in container confinement but also lends itself well to continuous security testing in modern software development environments. Because it is both safe and modular, Houdini can be seamlessly integrated into automated systems, such as CI/CD pipelines, to provide ongoing validation of container security.

Regression testing is a widely adopted practice in software engineering to ensure that updates or changes to code do not introduce unintended failures or security regressions. In the context of container security, integrating Houdini into CI/CD pipelines enables organizations to automatically trigger a comprehensive suite of confinement tests whenever any part of the container stack changes, such as kernel upgrades, Docker releases, or policy updates. This continuous validation ensures that every new deployment or system update is accompanied

by an up-to-date security assessment. By systematically rerunning empirical security tests across different software versions and configurations, *Houdini* helps detect configuration drift and regression in security guarantees that might otherwise go unnoticed, maintaining the effectiveness of container isolation mechanisms as platforms evolve.

Finally, to foster further research and encourage community adoption, we have made the source code for *Houdini* publicly available³, along with a base set of initial tricks. We hope this will enable others to extend *Houdini*'s capabilities, contribute new test cases, and collectively advance the state of container security benchmarking.

VIII. RELATED WORK

At the time of this writing, to our knowledge, *Houdini* is the first comprehensive approach for evaluating and comparing *container confinement mechanisms*. Much of the container security literature to date has focused on either vulnerability scanning of container images, building offensive/attack tools for container escapes, or proposing best practices for securing container runtimes. While these papers, tools, and documents do not directly share our research objectives, they still broadly fit into the container security landscape. Additionally, research on VM escapes and sandbox escapes has provided valuable insights into isolation vulnerabilities, many of which relate to container security. Techniques used to break out of hypervisors and sandboxed environments often share similarities with container escape methods, which the importance of strong confinement mechanisms. The remainder of this section does not aim to exhaustively enumerate all container security tools and systems, but rather to highlight salient methods, including prior research on VM and sandbox escapes.

Regression Testing. One common practice in continuous integration/continuous deployment (CI/CD) pipelines is the use of automated vulnerability scanning tools to detect known vulnerabilities in container images. Tools such as Trivy [8] and Clair [4] are often integrated into CI pipelines to perform vulnerability assessments during the image build process. These tools automatically scan for security flaws in the container image, ensuring that only secure images are deployed to production. For example, GitLab's CI/CD pipeline includes built-in features for Container Scanning with Trivy, automatically running vulnerability scans whenever new Docker images are built or updated.

In addition to security scanning and policy enforcement, CI/CD pipelines also use container hardening tools like Docker Bench for Security [9] and Kube-bench [10] to ensure containers are securely configured. These tools can be integrated into the pipeline to automatically check container configurations against security best practices, such as ensuring that containers run as non-root users or that unnecessary ports are closed. CircleCI [11], for example, integrates Docker Bench to verify that new container builds follow best practices before they are pushed to production.

Vulnerability Scanners. Many free, open, and closed-source tools exist for identifying the presence of known vulnerabilities in container images. Shu *et al.* [12] developed a vulnerability scanning tool to scan Docker Hub container images at scale. They found that on average, official and community images have concerning amounts of vulnerabilities (180+), and that these vulnerabilities remain unpatched for hundreds of days. Their study utilized the Docker Image Vulnerability Analysis (DIVA) framework, which automates the discovery, download, and analysis of over 300,000 Docker images. The findings revealed that more than 80% of both official and community images contained at least one high-severity vulnerability, and many images were not updated for extended periods. Moreover, vulnerabilities were often inherited from parent images to child images, further increasing security risks.

Best Practices. Users in search of (often high-level) advice on how to improve the security of their container deployments can consult one of many best practices guides available online. One such guide is the Center for Internet Security (CIS) Docker Benchmark [1]. In our experience, guides tend to offer generic advice (e.g., “Only allow read access to the root filesystem”). This is problematic because it oversimplifies container security. Generic advice may seem straightforward, but it doesn't account for the specific context or unique configuration of an individual deployment. Such one-size-fits-all recommendations can lead users to believe that simply following these broad guidelines is enough to secure their systems, potentially leaving critical vulnerabilities unaddressed. In reality, effective container security requires a nuanced, tailored approach that considers the particular needs and threat models of each environment. *Houdini* fills the gap between generic advice and practical security by enabling one to objectively evaluate the effectiveness of your container confinement measures.

Offensive Tools. The offensive security community has developed a range of specialized tools aimed at identifying and exploiting vulnerabilities within containerized environments. Containers, while providing strong isolation, are not immune to attacks, and these tools focus on testing the robustness of container defenses. CDK⁴ and DEEPCE⁵ are two widely used, open-source container penetration testing toolkits that leverage a collection of known exploits to gain persistence, escape the container, and gather sensitive information about the container environment. These tools aim to bypass isolation mechanisms like namespaces, cgroups, and SELinux, often using a variety of aggressive techniques to identify weaknesses in container security. CDK specifically uses any available method to circumvent security measures, providing a broad toolkit to test the limits of a container's defense. Meanwhile, DEEPCE, developed by stealthcopter, focuses on privilege escalation and container enumeration, attempting to exploit specific weaknesses to achieve container escapes, thereby compromising the host system. In contrast, *Houdini* takes a more systematic and structured approach to container security testing. Rather

³<https://github.com/CU-Security-Labs/houdini>

⁴<https://github.com/cdk-team/CDK>

⁵<https://github.com/stealthcopter/deepce>

than focusing solely on exploiting vulnerabilities, *Houdini* allows for the direct comparison of defensive techniques by clearly defining the testing environment, outlining the exploit steps, and documenting the expected outputs.

Virtual Machine Escapes. VM escapes occur when an attacker successfully breaks out of a virtualized confined environment, gaining unauthorized access to the host system. Given that hypervisors manage multiple VMs on shared hardware, such escapes can lead to host compromise, cross-VM attacks, and lateral movement within cloud environments. One of the most notable VM escape vulnerabilities is VENOM (CVE-2015-3456) [13], which exploited a flaw in the virtual floppy disk controller implementation in QEMU, affecting platforms like Xen, KVM, and VirtualBox. Other significant VM escape vulnerabilities include Cloudburst (2009), which exploited VMware’s virtual graphics driver; CVE-2018-6981 and CVE-2018-6982, which targeted VMware ESXi, Workstation, and Fusion [14]; and multiple VirtualBox vulnerabilities (2017) enabling guest-to-host code execution [15]. Additionally, side-channel attacks like MDS/ZombieLoad (2019) demonstrated how speculative execution flaws could be leveraged across VMs [16]. More recent research has uncovered weaknesses in encrypted virtualization, such as the SEVered attack (2018), which extracted plaintext memory from AMD SEV-protected VMs [17], and the Heckler attack (2024), which manipulated non-timer interrupts to compromise confidential VMs in AMD SEV-SNP and Intel TDX [18].

Sandbox Escapes. Sandboxing is a critical security mechanism for restricting the execution of untrusted code, commonly used in browsers, mobile apps, and security tools. A sandbox escape occurs when an attacker bypasses these restrictions, executing arbitrary code with higher privileges and potentially leading to full system compromise. Modern web browsers heavily rely on sandboxing to isolate JavaScript execution from the underlying OS. However, several high-profile vulnerabilities have enabled sandbox escapes. For example, CVE-2019-5786 [19], a use-after-free vulnerability in the FileReader API of Google Chrome, allowed attackers to escape Chrome’s sandbox and execute code on the host. Other notable sandbox escape vulnerabilities include CVE-2022-26706, which exploited a flaw in macOS’s App Sandbox to execute code outside of restricted permissions [20]; and CVE-2023-36719, a stack corruption vulnerability in Windows OS that enabled escape from the Chromium sandbox [21]. Additionally, vulnerabilities in Judge0, an online code execution system, exposed systems to full takeover by allowing attackers to escape its sandbox [22]. Research has also uncovered critical flaws in vm2, a JavaScript sandbox library, which could be exploited to bypass its security restrictions and gain remote code execution [23].

IX. CONCLUSION

In this paper, we presented *Houdini*, a security benchmarking framework designed to evaluate the effectiveness of container confinement mechanisms. As container-based workloads continue to dominate cloud infrastructures, ensuring that

containers are properly isolated is paramount for maintaining security. *Houdini* offers a systematic and empirical approach to test and assess various container security configurations, highlighting potential misconfigurations that may compromise container isolation. Our evaluation demonstrated the ability of *Houdini* to detect critical misconfigurations, including those leading to privilege escalation. By providing a reliable and reproducible method for testing container confinement, *Houdini* helps bridge the gap between theoretical security assumptions and practical real-world behavior.

As container technologies evolve, it is essential to continuously assess and improve their security mechanisms. *Houdini*’s modular design ensures that it remains adaptable to new security challenges and configurations, allowing for ongoing validation and refinement of container security practices. By offering an open framework for container confinement benchmarking, *Houdini* aims to support the development of more secure container technologies, ultimately contributing to a safer cloud infrastructure for the future.

REFERENCES

- [1] Cis docker benchmark. In *CIS Docker Benchmark*, 2024.
- [2] John L Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, 2000.
- [3] Raghunath Nambiar et al. Transaction Processing Performance Council (TPC): State of the Council 2010. In *Performance Evaluation, Measurement and Characterization of Complex Systems*, pages 1–9. Springer Berlin Heidelberg, 2011.
- [4] Quay. Clair: Vulnerability static analysis for containers. <https://github.com/quay/clair>, n.d.
- [5] William Findlay, David Barrera, and Anil Somayaji. Bpfcontain: Fixing the soft underbelly of container security. *arXiv preprint arXiv:2102.06972*, 2021.
- [6] Buildroot Developers. *Buildroot: Making Embedded Linux Easy*. Buildroot, 2024. Available at <https://buildroot.org>.
- [7] Open Containers Maintainers. runc: File descriptor leak leading to container breakout (CVE-2024-21626). <https://github.com/opencontainers/runc/security/advisories/GHSA-xr7r-f8xq-vfvv>, 2024. Accessed: 2024-04-28.
- [8] Aqua Security. Trivy: A simple and comprehensive vulnerability scanner for containers and other artifacts. <https://github.com/aquasecurity/trivy>.
- [9] Docker. Docker bench for security: A security testing tool for docker. <https://github.com/docker/docker-bench-security>, n.d.
- [10] Aqua Security. Kube-bench: Checks whether kubernetes is deployed securely. <https://github.com/aquasecurity/kube-bench>, n.d.
- [11] CircleCI. Continuous integration and delivery platform. <https://circleci.com>. Accessed on: June 23, 2025.
- [12] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280, 2017.
- [13] Cheick Abdoul-Kader and Shih-Hao Chang. A novel venom attack identification mechanism in cloud virtualization environment. *Communications of the CCISA*, 24(1):61–72, 2018.
- [14] VMware Security Team. VMware virtual machine escape vulnerabilities: Cve-2018-6981 and cve-2018-6982. 2018. Accessed on: March 3, 2025.
- [15] Google Project Zero. 10 new vm escape vulnerabilities discovered in virtualbox. 2017. Accessed on: March 3, 2025.
- [16] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.
- [17] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting amd’s virtual machine encryption. In *European Workshop on Systems Security (EuroSec)*, EuroSec’18, 2018.
- [18] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. {HECKLER}: Breaking confidential {VMs} with malicious interrupts. In *USENIX Security*, 2024.

- [19] Philippe Laulheret. Analysis of a chrome zero day: Cve-2019-5786, 2019. Accessed on: March 3, 2025.
- [20] Jonathan Bar Or. Uncovering a macOS App Sandbox Escape Vulnerability: A Deep Dive into CVE-2022-26706, 2022. Accessed on: March 3, 2025.
- [21] James Forshaw. Escaping the Sandbox: A Bug That Speaks for Itself, 2023. Accessed on: March 3, 2025.
- [22] Ravie Lakshmanan. Sandbox Escape Vulnerabilities in Judge0 Expose Systems to Complete Takeover, 2024. Accessed on: March 3, 2025.
- [23] FortiGuard Labs. Vm2 sandbox escape vulnerability — outbreak alert, 2023. Accessed on: March 3, 2025.