Abstract

In this paper we introduce TIMELOPS a novel technique for automatically learning system call filtering policies for containerized microservices applications. At run-time, TIMELOPS automatically learns which system calls a program should be allowed to invoke while rejecting attempts to call spurious system calls. Further, TIMELOPS addresses many of the shortcomings of state-of-the-art static analysis-based techniques, such as the ability to generate tight filters for programs written in interpreted languages such as PHP, Python, and JavaScript. TIMELOPS has a simple and robust implementation because it is mainly built out of commodity, and proven, technologies such as seccomp-BPF, systemd, and Podman containers, with fewer than 500 lines of code. We demonstrate the utility of TIMELOPS by learning system calls for individual services and two microservices benchmark applications, which utilize popular technologies like Python Flask, Nginx (with PHP and Lua modules), Apache Thrift, Memcached, Redis, and MongoDB. Further, the amortized performance of TIMELOPS is similar to that of an unhardened system while producing a smaller system call filter than state-of-the-art static analysis-based techniques.

1 Introduction

Microservices have become very prominent due to shifts from monolithic to modularized and distributed architectures. Many large companies such as Netflix, Twitter, and Amazon are heavily reliant on microservices to create scalable applications [9] [17]. This paradigm for computing however, introduces many complexities such as load-balancing, scalability, and our biggest concern, security. Microservices applications are constantly evolving and new nodes are incessantly created and destroyed [22]. Because of this, it is difficult to thoroughly understand the behavior of each service at any point in time in time and manually write appropriate security policies. Restricting services to only behave safely as the developer intended is challenging for microservices, but especially necessary since they typically run on a shared host with other multitenant cloud services. Malicious actors performing privilege escalation attacks or container escapes on a shared host could have catastrophic effects on many users.

System call filtering is a widely deployed security technique in the cloud [23, 27]. System call filtering enables the principle of least privilege [41] by restricting system calls available to a program to minimize the amount of interaction that a compromised application may have with the kernel and system resources. Because of the nature of microservices, they are excellent candidates for system call filtering. Since containers in microservices applications contain small lightweight services, applying custom system call filters to each service is a great opportunity to apply fine-grained security policies. Further, the multitenant nature of these deployed services incentivizes attackers to induce remote code execution via vulnerabilities in user code or shared libraries. Violating the container isolation boundaries is often achieved through calls to the system call library, which we aim to limit via system call filtering. This security feature would not only be beneficial to cloud providers protecting their host machines but also customers that want to ensure the integrity of their application and data regardless of which applications are scheduled to run on the same physical host as their application.

The efficacy of system call filtering mechanism is dependent on the actual policy used with the filtering mechanism. Deriving a policy, viz., figuring out which system calls should be permitted is generally a difficult, delicate balancing act for security engineers: including unnecessary system calls in the filter can permit security attacks, while excluding necessary system calls can result in unnecessary alarms or spurious program termination.

In the early days of system call filtering, candidates for filtering were chosen based on oblivious system features (e.g., 32-bit calls on a 64-bit machines) or supplied by programmers based on their understanding of their code [2, 12]. To expand the scope of system call filtering, recently, researchers have used sophisticated static analysis with great success: for instance, recent works [15, 19, 24], demonstrate...
that for C/C++ applications the system call set can be very close to the set used at run-time.

One limitation of current static analysis solutions is that they cannot be used for microservices written in interpreted languages such as JavaScript, Python, PHP, however, this is the way many microservices are written [13]. Frameworks in these interpreted languages allow programmers to quickly stand up services that offer a plethora of features such as hosting web servers, communicating with databases, or allowing web authentication [7] [4] [6]. Static analysis is not useful for such deployments because the analysis can only analyze the interpreter, which, by definition, should be able to support all possible system calls that can be executed by a program in that language. An alternative solution is to analyze the source code of the interpreted language and then map function/library calls in the language to system calls [30]. In our method, system calls are learned, on-the-fly, as programs execute, thereby including only system calls that are actually observed. The challenge here is to detect and reject malicious system calls that can be encountered in the learning phase from being added to the system call list.

The TIMELOOPS technique relies on three components for learning legitimate system calls: (1) a production service, (2) an oracle service, and (3) a TIMELOOPS controller. The production service is an unhardened service that is typically tuned for high-performance, while the oracle service is a hardened service that is typically tuned for high security (at the cost of performance). The high level idea is to sparingly use the oracle service to decide if a system call should be added to the allow list. This provides the security guarantees of a hardening technique without the overheads associated with hardening.

We start execution with our production service and an empty system call filter. When the TIMELOOPS controller encounters a system call that is not already in the system call filter, the controller kills the production service and stands up the oracle service. Since cloud microservices are typically designed to be stateless, we can take advantage of retry semantics where clients can resend requests until they receive the appropriate response. When the oracle service is started, we resend the the request that triggered this system call violation but this time, it is received by our oracle service. As per our assumption, the oracle service will fail upon malicious inputs. However, if the oracle processes the input it will also encounter the same system call that killed the production service. When this happens, the TIMELOOPS controller updates the system call filter and switches back to the production service. There are several hardening techniques that can be used for the oracle to detect malicious behavior such as ASan [42], EffectiveSan [21], SGX [49], isolated execution for micro-architectural safety, etc.

We demonstrate TIMELOOPS by implementing our idea(s) in an environment representative of cloud workloads today, using technologies that are commercially available today. We use seccomp-BPF [45] for system call enforcement, systemd [5] to implement orchestration services in our TIMELOOPS controller, and Podman [40], a rootless container service, to isolate and restart the production and oracle services. For the hardened version of the application we use AddressSanitizer [42]. The glue logic for TIMELOOPS system is roughly 500 Python LoC. We show how the idea can be implemented for Nginx with PHP modules and Lua, a Python Flask service, and two benchmark microservices applications. Nginx, PHP, Lua, and Flask are commonly used in developing microservices so they were suitable for evaluating Timeloops. This infrastructure is compatible with the Podman and Docker interfaces and can be refactored to be integrated with other container technologies. Because microservices are stateless and can be stopped and restarted without disturbing the correctness of the application, they are compatible with our design.

Using these components we measure both the security and performance of our TIMELOOPS system. We observe that the amortized performance of the TIMELOOPS system is similar to an unhardened, insecure system, and our system nearly eliminates all the performance overheads of the hardened system. In terms of the tightness of the system call filtering set, we show that a state-of-the-art static analysis tool generated system call policies that were 32.7% larger than the ones generated by TIMELOOPS. Additionally, we demonstrate that TIMELOOPS succeeds in creating system call filters for interpreted languages with very little effort, which is a challenge for existing static analysis-based techniques.

Our contributions are as follows:

1. We propose a novel technique for dynamically learning security policies with minimal run-time performance costs.

2. We present the TIMELOOPS system, which applies this technique to learning system call policies while detecting memory corruption exploits.

3. We evaluate the TIMELOOPS system for performance and compare its generated policies to current state-of-the-art automated system call policy generating tools.

The remainder of this paper is organized as follows: in Section 2, we provide the necessary background information and present our threat model. In Section 3, we introduce a novel approach for learning new security policies that leverages run-time exploit detection techniques while providing
amortized run-time performance overhead. In Section 4, we introduce the TIMELOOPS system, a concrete application of the approach to learn flexible system call policies for containerized applications by using an ASan oracle to detect memory corruption attempts. We consider the security architecture of the TIMELOOPS system in Section 5, and evaluate the system’s run-time performance and syscall filter learning correctness in Section 6. We discuss related work in Section 7, and conclude our work in Section 9.

2 Background and Threat Model

2.1 System Call Policy Generation

The system call (syscall) interface enables user-space applications to request privileged services from OS kernel. This interface is quite large: for example, the Linux kernel (v5.x) provides \(\approx350\) syscalls to user-space applications [19]. Most programs only need a subset of these syscalls for proper execution [15, 19, 23, 24, 27]. An attacker, however, who is able to leverage a vulnerability in a userland application to gain code execution can use any of the available syscalls, effectively (1) violating the principle of least privilege or (2) further (ab)using vulnerabilities in less-stressed kernel code paths to escalate privilege [31]. By restricting the system calls available to a process, an attacker is constrained to only performing actions that fall within the benign behavior(s) of the victim program.

Determining which system calls should be allowed for a program’s execution is challenging. Policies can be created manually, but with significant programmer effort [20]. Therefore, attempts to synthesize policies automatically have become more prevalent [15, 19, 23, 24, 27]. Current approaches for automatically generating system call policies oftentimes use either static or dynamic analyses (or a combination thereof). Static analyses often attempt to reason about the system calls that a target program should be allowed to execute without executing the program [19, 23, 24]. These approaches are generally over-approximate in their analyses: they are able to identify all system calls that a program must be able to execute, but also identify system calls that the program does not require.

Dynamic analyses typically execute the target program with some set of known-safe training inputs, like developer-written tests, in order to record all executed system calls and to populate an allow-list from them [15, 27]. These approaches are generally under-approximate, because they are only able to identify system calls that are observed during execution of training data, and fail when a new system call is executed, even if the system call should have been allowed.

For both static and dynamic system call policy extraction, current approaches create immutable policies that cannot be later refined.

For example, consider the case of a dynamic analysis-based syscall extraction scheme that creates an allow-list from observed system calls while executing a series of developer-written tests: once trained, the target program is executed with the allow-list enforced. In an immutable system, when the target program attempts to execute a non-allowed system call, the enforcing mechanism will terminate the program in order to prevent potential exploitation. This occurs at the risk of the system call being the result of a safe input that was incorrectly omitted from the training data. In order for it to be flexible enough to update its policy once deployed, the system must be able to distinguish the execution of syscalls that result from benign inputs from the syscalls that result from potentially malicious inputs.

2.2 System Call Policy Enforcement

Once a system call policy is created in the form of an allow-or deny-list, it must be enforced. The Linux kernel provides the Secure Computing Mode (seccomp) that allows users to restrict the system calls that a process is allowed to make [34]. Originally introduced as a mechanism for restricting a calling thread to only making calls to `read()`, `write()`, `_exit()`, and `sigreturn()` and no other system calls, seccomp was later extended to allow for defining custom system call filters written as BSD Packet Filters (BPF) [35]. seccomp-BPF allows a user to configure the behavior of the kernel when the user-space process makes a non-allowed system call; for example, the kernel can be configured to send a SIGKILL or SIGSYS signal to the process, depending on the arguments provided to via the seccomp interface, log the event to an audit log, etc. [45].

2.3 Run-time Exploit Detection and Mitigation

In order for an attacker to gain arbitrary code execution and execute system calls for nefarious purposes, they often must exploit a vulnerability or misconfiguration in the corresponding application. To prevent exploitation, a wide range of research proposals aims at hardening application code using various run-time verification and enforcement mechanisms [14, 29, 43, 44]. Many of these prevent the exploitation of entire vulnerability classes in exchange for additional complexity in application code that results in a run-time performance cost. The decision to deploy any given technique becomes a tradeoff between the security benefits and the performance cost [11, 44].

For example, AddressSanitizer (ASan) [42] provides detection of memory-safety-related issues (e.g., out-of-bounds accesses and uses of freed heap objects), but with reported run-time overheads of 73% when first introduced. Because of the high performance overhead, ASan has primarily been employed, to good effect, in security auditing and testing environments and not for production deployments.
In our implementation of the TIMELOVES system presented in Section 4, we demonstrate an approach for applying runtime exploit detection and mitigation techniques in production environments in a manner that amortizes their run-time performance costs. This enables the use of techniques that are too slow to be seriously considered for production environments, but with the new consideration for system calls the technique may introduce to the analysis.

2.4 Container Security

Containers provide process isolation from host resources by leveraging OS-provided interfaces, like namespaces [37] and control groups (cgroups) [33]. This allows users to execute applications in isolated environments that have their own, unique view of system resources, such as mount points, process and user IDs, network stacks, IPC objects, etc. [39]. The shift toward deploying applications as microservices in public and private cloud environments has embraced containerization and promoted multitenant deployments of applications where multiple isolated applications execute on the same host [30].

Attackers who are able to execute code in applications running in containers are limited in their ability to interact with system resources because of the isolation provided by the containerized environment. However, attackers can sometimes "escape" from the containerized environment by exploiting vulnerabilities or misconfigurations in the container to remove the host isolation, gain access to system resources [16], and possibly elevate user privileges. Escaping from a container in a multitenant environment could provide an attacker with access to any other container executing on the system. Container escapes can occur when specific system calls are allowed to be executed that break the isolation between the host and the container [31]. Because the container shares the same kernel with all other host applications, any system call executed by the kernel on behalf of the container application runs the risk of affecting system resources that maintain the container isolation.

To prevent container escapes, container environments expose a seccomp-BPF interface to restrict the system calls available to a container application [23, 27]. Common container implementations (e.g., Docker, Podman) also provide a sane default seccomp filter that removes access to 44 system calls for all containers. The container seccomp-BPF interface can be leveraged to further restrict the system calls available to an application executing in a container [20].

The attacker is able to exploit some vulnerability or misconfiguration to execute code in the container process [44], and may desire to escape the container environment or compromise the container application for malicious purposes via executing arbitrary system calls [11, 16, 31]. Our threat model is in par with prior work in the area [15, 19, 24].

3 Timelovess Security Policy Learning

For a system to be able to learn a security policy at run-time, it must be able to determine whether a violation of the current system is the result of a benign input that was previously unseen or the result of a malicious input that should be prevented by the policy. In this section, we describe a novel approach for learning software security policies by introducing an oracle component to assist with determinations of safe inputs.

Our policy learning approach provides a mechanism to learn from policy violations. When a violation occurs after the program executes a specific input, we consult the oracle service by providing it the current policy and input and re-execute the program under the oracle’s observation. The oracle makes a determination that the security policy violation occurred either because of a benign input to the program or because of a malicious input to the program. If the oracle determines that the policy violation was due to a benign input, then the policy is updated to allow the event that was observed, and the program can be restarted with the updated policy. If the oracle determines that the policy violation was due to a malicious input, then an exploit attempt was prevented and an alert is raised accordingly.

For this approach to succeed, the program must be resilient to repeated re-executions from a known state, whether from the program start or from some checkpoint location. We observe that network services, and in particular, microservices, are often designed to be stateless and support the re-execution of idempotent request operations. Because of this, we can take advantage of retry semantics or we can record and re-play requests without affecting the correctness of the entire application’s state. For engineering simplicity, we assume that retry semantics exist and our application’s clients resend requests until they succeed. These types of services can be re-executed and are thus good candidates for a timelooping security policy learning approach.

4 Learning System Call Policies

In this section, we describe the design and implementation of the TIMELOVES system, which we built to dynamically learn system call policies for securing container applications. We set out to demonstrate that system call policies can be learned dynamically by querying an oracle service to determine whether the syscall violation should be allowed.
4.1 Designing TIMELOOPS for Learning System Call Policies

The TIMELOOPS system is designed as three main components: (1) the application, (2) the oracle, and (3) the controller service. Both the application and oracle are deployed in their own isolated container environments, while the controller service is responsible for managing the execution of the application and oracle containers and for applying policy changes as information is learned.

The TIMELOOPS controller begins execution by starting the application container with some default system call policy of allowed syscalls (which may be initialized as an empty list). If, given some input, the application container attempts to execute a syscall that is not in the allow-list, the application is forced to exit and the controller is notified. The controller then starts the oracle container, which starts the same application program but with additional run-time exploit detection techniques applied; for example, the application program can be compiled with AddressSanitizer (ASan) to act as an oracle for some types of memory corruption exploits. Importantly, the controller service starts the oracle service with the same allow-list of system calls that was applied to the application. Given the same input, the oracle should either (1) exit due to the same system call violation, or (2) exit due to an ASan abort. Case (1) indicates that the code path taken to reach the system call did not result in memory corruption, and so the offending system call may safely be added to the allow-list — in this case, the controller service updates the policy and starts the application with the new policy. Case (2) indicates that prior to reaching the new system call, a memory corruption violation occurred, and that the input is unsafe.

To summarize, TIMELOOPS identifies when a new system call violates an existing policy, queries an oracle to determine whether the input that caused the system call execution is “safe” according to the run-time exploitation detection features of the oracle, and makes policy update decisions based on the determination. An additional benefit is that the run-time performance overhead of executing a program with exploit detection and mitigations applied occurs lazily, i.e., only when a system call policy violation occurs. This means that in the critical path of the program execution, the overall system executes with the amortized run-time of the application, while still providing the benefits of executing exploit detection checks opportunistically. We show the evaluated run-time performance of TIMELOOPS in Section 6.

4.2 Container Implementation

The application and oracle programs are isolated in their own container environments. This provides separation from system resources and from the controller service that manages the security policy. We chose to use Podman containers [3] in the TIMELOOPS implementation. Podman is an alternate to Docker, the most popular container engine. Podman’s primary advantage is its daemonless architecture — unlike Docker, which uses a single daemon executing with root privileges to manage containers, Podman can create containers as non-root child processes. Podman is compliant with the Open-Container Initiative (OCI) standard, which describes an interface between container front-ends, like Docker and Podman, and the back-end container runtime. The TIMELOOPS controller leverages these features to start and stop the application and oracle containers as systemd services. Additionally, by using containers, we make our TIMELOOPS system convenient for adoption in stateless microservice deployments.

4.3 System Call Filtering in Containers

Both Podman and Docker provide support for applying a seccomp-BPF filter to a container on start-up. This functionality is enabled by default and restricts the system calls available to an executing container. The default filter [10] removes access to at least 44 system calls that can be abused for container escape exploits, but which still leaves over 300 system calls available.

We use the Podman feature to apply a custom seccomp-BPF filter every time the TIMELOOPS controller starts the application and oracle containers. In between executions of the containers, the controller may update the filter based on information learned from the oracle, but once a container is started with a filter, the filter (and therefore, the set of syscalls that the container is allowed to execute) is immutable until the container exits.

4.4 Applying Run-Time Exploit Detection to the ASan Oracle

In our implementation, we use AddressSanitizer (ASan) as the run-time exploit detection technique for the oracle. ASan is provided as compiler-inserted instrumentation for detecting run-time memory violations like out-of-bounds accesses and use-after-free bugs, but at the cost of run-time performance overheads. To create an oracle container for a given application with ASan applied, a user must compile the application code with ASan compilation flags — we chose to use Clang’s ASan support for our applications. Oracle containers can be created using other exploit detection techniques to detect other types of vulnerabilities specific to an application.

4.5 Handling Concurrent Applications

In order to handle concurrency in applications with multiple threads or processes, TIMELOOPS needs to ensure that a system call policy violation in any child thread results in the container exit, i.e., the init process of the container terminates. The seccomp-BPF interface provided by Podman allows for configuring the kernel response to a non-allowed system calls.
The default configuration options provided by Podman were limited, but in most cases, selecting the configuration to send a SIGSYS signal to the entire thread group resulted in the desired behavior.

Consider, however, the case of a forking server where a main process forks multiple child processes to handle requests. A system call violation in one of the child processes results in a SIGSYS signal that terminates the child process, but the parent process (container init process) continues executing. In order to handle this case, a signal handler for the SIGSYS signal needs to be registered in the application and oracle containers by pre-loading it into all executing processes in the container. When SIGSYS is received by any process in the container, the signal handler will pass the signal to its parent process to cause it to terminate, until the init process is reached and the container exits. To ensure that this event is logged to the system’s audit log, we modified the crun container runtime program to change its default behavior to log SIGSYS trap events (by setting the SECCOMP_FILTER_FLAG_LOG flag as default) before recompiling it and configuring Podman to use our custom compiled crun binary. We intend to submit a pull request to the upstream Podman codebase to propose extending the existing seccomp-BPF API to additionally enable setting filter flags at container run-time, without requiring source code modifications to container run-time binaries.

4.6 System Calls Introduced by the Oracle

It is possible for the oracle container to execute system calls that the original application did not, which occurs when the exploit detection mechanism of the oracle introduces system calls that were not already required by the application. Therefore, a slight modification to the system as-described in Section 4.1 is required: add any system call that the oracle encounters during execution to the allow-list, assuming that the oracle does not detect exploitation prior to the system call being executed. These system calls can be trusted because the oracle is the determinant of safe inputs. However, introducing these syscalls potentially causes the allow-list generated to be a looser system call policy, so ideally we want to choose a hardening technique that provides the security guarantees we desire without introducing too many new system calls.

4.7 Retry Request Expectations

In the current TIMELoops implementation, the application may receive an input that results in a system call violation, causing the service to exit without handling the request. TIMELoops restarts the application container in the oracle and waits for the next request. We expect the client to use retry semantics, meaning that the client retries the request until successful. An alternative to this approach is for the TIMELoops controller to record each request and to replay it to the oracle, but for engineering simplicity while evaluating the approach, we assume that the client uses retry semantics.

Expecting the client to retry a request until a response is received may appear to be contrived, especially since we make no assumption that the client is benign or uncompromised. However, it does reflect the behavior of clients interacting with idempotent microservices, which, for robustness and fault tol-
erance, often expect retry semantics until a response confirms success [18]. Additionally, the expectation that identical messages are repeated does not introduce security vulnerabilities in the TIMELOOPS system when the expectation is not met — it only makes the system inefficient. The security implications of this are discussed further in Section 5. In the future, modifying the TIMELOOPS controller to record the most recently received request and to replay it to the oracle container in the event of a system call violation would remove this expectation without compromising significantly on performance.

### 4.8 Oracle Limitations on Policy Learning

The TIMELOOPS system is limited by the oracle container’s ability to detect exploit types. For example, the system as-described so far is able to determine when a new system call is the result of a memory corruption exploit that ASan can detect. However, when a system call is introduced by an attacker via an exploit or misconfiguration that does not leverage a memory corruption vulnerability, TIMELOOPS will update the system call policy to allow the new system call. Therefore, the scope of the security policies that TIMELOOPS can learn is limited by the abilities of the oracle. In Section 5.1, we discuss complimentary approaches for maintaining the security of the application container and the overall TIMELOOPS system in the context of exploits beyond what the oracle can detect. We also intend for the TIMELOOPS oracle model to be modular, such that multiple oracle containers with different forms of analyses can be deployed in parallel.

The sequence of events that occur when an application running in TIMELOOPS receives an input that causes a system call violation is illustrated in Figure 1.

### 5 Security Analysis

In this section, we consider the security features of the TIMELOOPS system and present the argument that it provides protection to a container application given reasonable assumptions about the threat model.

#### 5.1 Considering Possible Attacks

Given the presence of a remote attacker as described in our Threat Model (Section 2.5), four possible attack categories exist: (1) a malicious input that leverages a vulnerability that is detectable by the oracle (e.g., a memory corruption that ASan can detect) to execute a system call that is not allowed by the current policy; (2) a malicious input that uses an oracle-detectable vulnerability but does not execute any non-allowed system calls; (3) a malicious input that uses an oracle-undetectable vulnerability (e.g., a logic bug or misconfiguration) but does not execute any non-allowed system calls; and (4) a malicious input that uses an oracle-undetectable vulnerability to execute a system call that is non-allowed. Our TIMELOOPS implementation is designed to provide better defense against categories (1), (2), and (3) attacks than previous system call learning approaches, but we argue that mitigations can be applied that sufficiently prevent an attacker from compromising the system in the event of an attack of category (4). Categories (2) and (3) are functionally equivalent in terms of behavior and security of the TIMELOOPS system, which we will show.

Category (1) attacks (oracle-detectable exploit that executes a policy-violation system call) are detected and prevented by the TIMELOOPS system, and TIMELOOPS provides the strongest security benefits against these attacks by entirely preventing these kinds of attacks from gaining code execution in the application container. When the application container violates the system call policy as a result of the same malicious input, the oracle container will detect that the input is an exploit and prevent any changes to the security policy while alerting an administrator.

Category (2) attacks (oracle-detectable exploit that does not execute a policy-violation system call) and category (3) attacks (oracle-undetectable exploit that does not execute a policy-violation system call) are handled in the same way by the TIMELOOPS system and provide strong security benefits in the form of policy enforcement. Because the system call policy is never violated, the TIMELOOPS system is never alerted to the potentially malicious input and so the oracle is not given an opportunity to determine maliciousness. However, because of the tight system call policy applied to the application container, the attacker is restricted to being able to use only the system calls learned so far by the security policy, thereby limiting the attempts of the attacker to compromise the application or escape from the container. This is the same security benefit provided by other system call policy learning and sandboxing techniques, but with the intended benefit of providing a tighter filter on allowed system calls.

Category (4) attacks (oracle-undetectable exploit that executes a policy-violation system call) are most dangerous to the TIMELOOPS system, because, if not mitigated, TIMELOOPS is unable to prevent the execution of arbitrary system calls. When an input from this attack category is received, the system call policy violation will cause the application to exit, and for the oracle to execute. The oracle will not detect the maliciousness of the input, resulting in the TIMELOOPS controller adding the system call to the policy and restarting the application container. In this manner, an attacker can arbitrarily add any system calls to the security policy until a sufficient set are available to conduct the desired post-exploitation actions. Therefore, to mitigate the chance of this occurring, we propose two best practices when deploying an application with TIMELOOPS. First, careful consideration should be applied to the type of application and the vulnerabilities common to it in order to select the best oracle for the application type. Second, over-approximate system call filtering techniques can be applied to build policies that never contain system calls that are
known to be unnecessary. For example, the default Docker and Podman configuration applies a seccomp filter that removes access to 44 system calls that can be abused for container escapes. Further, a static analysis system call learning approach could be used to generate an over-approximation of the system calls that the application requires, plus any additions from the oracle. The TIMELOOPS controller can then check that any system call learned during execution also falls in the over-approximate set that was generated statically. In this way, static approaches can complement the TIMELOOPS approach to provide security against category (4) attacks that is as good as if using just the static approach, while TIMELOOPS provides stronger security benefits against category (1), (2), and (3) attacks.

5.2 Violating Retry Expectations

The analysis conducted in Section 5.1 expected that identical requests are repeated until a valid response is sent from the application. However, we must also consider the case where an attacker violates this expectation or sends a different malicious request instead of resending the original input. If this happens, then the oracle and the production services will each process different inputs, but we show that this does not compromise the security guarantees of our system.

The expectation that the client retry a request does not introduce any opportunities to violate security properties of the system. Consider the scenario where the attacker sends a request that results in a system call policy violation, causing the application container to exit and the oracle container to start. The attacker can then send a new input, which can either cause a syscall policy violation or not. In the case that the input introduces a new syscall, the oracle will still make a determination of whether to add the syscall to the policy based on whether it detects malicious behavior, so even though the input does not match the input received by the application, the oracle can still safely make determinations for adjusting the system call policy. In the case that the new input does not introduce a new syscall, the oracle will either detect exploitation (e.g., because the input causes memory corruption), or the input does not cause the oracle to exit. In either case, the security policy does not change and the safety of the system remains intact.

This scenario still allows an attacker to abuse the system architecture to cause targeted increased run-time overheads, despite the system call policy remaining intact. The attacker can induce repeated switches from application container to oracle by providing an input that results in a new system call, but by then providing inputs that do not cause system call violations, the oracle container handles all new requests but with the higher performance overhead that comes from the exploit detection instrumentation in the oracle. To prevent this from occurring (even accidentally), we add a watchdog timer to the oracle container that causes it to exit after some configurable amount of time, causing a switch back to the more performant application container.

5.3 Application Container Compromise

In the event that an attacker does achieve remote code execution in the application container, the TIMELOOPS system is engineered in a manner that prevents full-system compromise. First, the TIMELOOPS leverages container abstractions provided by Podman to isolate the application and oracle container resources. Further, the containers are restricted from accessing the full set of system calls by the container seccomp policy that is set by TIMELOOPS. The file representing the system call policy itself is maintained entirely outside of the executing container environments, and only passed as arguments to the Podman command line interface at container startup. Therefore, an attacker with code execution inside either container is unable to directly manipulate the policy file. Additionally, seccomp configurations affect kernel behavior, i.e., how the kernel behaves when the process attempts to execute a system call. All system call violations will result in SIGSYS signals delivered to the offending thread in the container, as well as records in the system audit log. It is possible for the attacker to manipulate the userspace handling of the SIGSYS signals, given code execution in the context of an application process. While this would prevent TIMELOOPS from functioning correctly, it would not allow the attacker to loosen the existing system call policy which is enforced by the kernel.

We further mitigate the danger of a successful remote attacker in our design. We leverage features of the Podman container engine to ensure that containers are not running with root privileges on the host system, so that any container escapes result in an attacker having the same permissions as the TIMELOOPS user, and no more. However, we believe that because of the series of security benefits provided by the TIMELOOPS system, from exploit detection, tight system call filtering, and container isolation, that TIMELOOPS sufficiently raises the bar for successful system compromise to be impractical.

5.4 Comparison to Permanent Hardening Techniques

We must also compare the security guarantees that TIMELOOPS provides with a permanently hardened system, or in our case, a corresponding service that is always running with ASan. In the TIMELOOPS design, we only check the inputs that trigger new system calls with ASan when we query the oracle. In the case that a malicious input does not invoke a new system call violation but induces memory corruption, a permanently hardened service will offer more security benefits with a significant performance cost. We offer less protection than an ASan-hardened service when the attacker
abides by the syscall policy, but we offer as much protection as an ASan-hardened service when the attacker does not. Malicious inputs tend to invoke new system calls, so by design, TIMELOOPS will inspect those inputs with greater scrutiny in our hardened oracle environment. Our approach provides the benefit of significantly reduced run-time overheads that make deployment in production settings more practical than a permanently hardened application.

6 Evaluation

We evaluate the TIMELOOPS system call learning implementation to answer the following questions:

1. **Performance**: How does TIMELOOPS affect the latency of responses for applications?

2. **System call filter correctness**: How well does TIMELOOPS generate correct system call allow-lists while preventing exploitable conditions in comparison to existing techniques?

6.1 Applications Evaluated

We conduct the analysis by executing four different applications in Timeloops. Two applications are statically served web sites hosted by Python Flask and Nginx, and two applications are the social network and media microservices benchmarks from the DeathStarBench suite [22].

6.1.1 Python Flask Application

Flask is a popular Python web development framework. We developed a simple Flask application that serves a static web site over HTTP connections to evaluate the TIMELOOPS system. Flask applications are executed by the Python interpreter and provide multi-threaded functionality. Many web applications are developed using similar frameworks using interpreted languages, like PHP, Ruby, and JavaScript. Since Python and its modules enable easily developing web applications or querying databases, it is useful for the quick development of microservices, and thus has become a popular choice for cloud developers and a suitable application to evaluate TIMELOOPS [26].

6.1.2 Nginx and PHP Web Site

Nginx is a free and open-source web server that is used by 32.9% of all web sites, the largest of any available web server [46] We built a simple, static website hosted by an Nginx server with PHP-FPM integrations, where PHP is used by 78.1% of web sites where the server-side language is known [47]. Both Nginx and PHP-FPM use a forking server model that forks worker processes to handle connections. We evaluate TIMELOOPS with this application to represent services that use multiple processes to provide web content. Additionally, the PHP modules used in this Nginx application rely on an interpreter which makes predicting its behavior difficult, but suitable for our application like the Python Flask application.

6.1.3 Microservices Benchmarks

Containerized applications are frequently used in microservice cloud deployments and distributed computing settings. In order to evaluate TIMELOOPS in an environment representative of real-world multi-service applications, we selected microservice applications from the DeathStarBench benchmark suite of microservices [22]. We specifically evaluated against the media benchmark, which models the structure of popular media streaming services like Netflix, and the social network benchmark, which models a social network site like Twitter. Both applications depend on technologies used in real microservice deployments such as Memcached, Redis, Thrift, MongoDB, and Nginx, and its services are written in a variety of programming languages.

To deploy these services, we ran each microservice from a benchmark on a different Amazon EC2 instance [8] as it might be deployed in practice. Each EC2 instance had its own TIMELOOPS controller setup and generated a unique filter for each service. We made minor changes to the microservices and added more robust error handling.

6.2 Performance

To evaluate the performance of TIMELOOPS, we consider the end-to-end latency of network requests sent to a service application running in the TIMELOOPS system. We measure latency as the elapsed time between a client making the first request to the service and the client receiving a response, regardless of the number of times the client may have retried the request.

We compare the latency of the application running in the TIMELOOPS system to the latency of the same application running with no additional exploit mitigations applied, i.e., not running in the TIMELOOPS system and compiled without default protections. This represents a baseline service that has no ability to detect violations of security boundaries, but which should execute with the lowest latency. We additionally compare the latency of the TIMELOOPS application to that of the same application executing on a non-TIMELOOPS system but compiled with AddressSanitization (ASan) applied. This represents a system that is instrumented to maximally determine when an input causes a security violation, but with a heavy performance cost as a result of the instrumentation. Our evaluation aims to show that TIMELOOPS can monitor for violations of security boundaries while also providing amortized performance benefits that are comparable to that of an undefended system.
Figure 2: Comparison of cumulative time for processing repeated requests sent to an unhardened service, a TIMELOOPS service, and a hardened service. Figures 2(a) – 2(d) show the end-to-end latency of sending requests to four different types of requests that are in the media microservices (MM) and social network microservices (SN) benchmark. Figures 2(e) and 2(f) show the end-to-end latency of sending requests to each of the stand-alone applications that process web requests. In every case, the TIMELOOPS service starts with a longer response time while system calls are learned, but eventually overtakes the hardened service (intercept is plotted).

Figure 3: Comparison of request latency distribution per request sent to an unhardened service, a TIMELOOPS service, and a hardened service. The average request latencies of the TIMELOOPS services are similar to that of the unhardened service and lower than the hardened service, despite the maximum value of the TIMELOOPS service being a significant outlier. The maximum value comes from the large delay in servicing the first requests as system calls are learned.
All experiments began by executing the test application service in TIMELOOPS with an empty system call allow-list. This forced TIMELOOPS to learn the minimum initial set of system calls that are required by the test application before any inputs could be processed. Once TIMELOOPS had learned enough system calls such that the application was ready to receive inputs, we used an HTTP client program to begin sending identical repeated requests to the test application, which resulted in new system calls being learned on the first occurrence of an input being processed.

In all experiments, the very first request of a given input sent to the TIMELOOPS application resulted in very slow response times as the application encountered previously, new and unseen system calls and tested for their validity in the oracle container. Figure 2 shows the latency of each request sent, and of how the application running in TIMELOOPS always has a slower start relative to other two services. However, the average latency, shown in Figure 3, of every ensuing request processed by the TIMELOOPS application was closer to that of the completely unhardened service, which resulted in the TIMELOOPS service quickly outperforming the hardened service.

The high latency of learning a system call for a safe input only occurs the first time that the system call is executed. Once a system call is learned from an input, all future occurrences of that system call will avoid the latency performance cost. Therefore, by pre-training TIMELOOPS on known-safe inputs to the application, an initial allow-list can be created prior to deployment. However, unlike in other traditional dynamic system call learning systems, these initial inputs do not need to exhaustively include every system call that should be learned — in this case, the TIMELOOPS system can still learn new system calls from future inputs, with a performance trade-off. By training on the most performance-critical inputs ahead of time, an operator of the TIMELOOPS system can further amortize system call learning costs on the critical path.

6.3 System Call Policies

In addition to performance, we evaluated the quality of the system call policies created by TIMELOOPS. To determine the effectiveness of TIMELOOPS for generating correct and safe policies, we sought to determine the following:

1. How many fewer system calls does TIMELOOPS allow than over-approximate static analysis techniques?
2. Which system calls do static analysis techniques allow that are not allowed by TIMELOOPS?
3. Which system calls does TIMELOOPS allow that are not allowed by static analysis techniques?
4. Which system calls does TIMELOOPS allow that would not have otherwise be necessary for the application to execute safely in a non-TIMELOOPS system?

We chose sysfilter [19] to represent state-of-the-art static analysis techniques for system call policy creation. As a static analysis tool, sysfilter tends to be over-approximate in its determinations, but sound. We used sysfilter to generate system call profiles that contain allowed system calls for all valid inputs to the evaluated program.

We generated system call policies for each evaluated program by executing it in the TIMELOOPS system and provided known-valid inputs to the program that exercised the correct functionality of the service.

We additionally profiled the behavior of each evaluated program when executed without TIMELOOPS by executing it in a container with seccomp RET_LOG mode, which executes and logs each system call. We provided a set of sample inputs to the profiled service and monitored the system calls invoked. This technique can be considered a naive dynamic analysis technique for generating system call filters. This provided a baseline of system calls that the program executes for a given input without the introduction of new system calls from the TIMELOOPS system.

For all evaluated programs, sysfilter created policies that were at least as large as those created by TIMELOOPS. On average, the sysfilter generated policies were 32.7% larger than those generated by TIMELOOPS, meaning that an attacker has access to approximately 32.7% more system calls when exploiting a system defended by sysfilter over one defended by TIMELOOPS, resulting in a larger attack surface. Static approaches ignore the fact that inputs modulate program behavior, and estimate the system call behavior solely based on the program and thus over-approximate the set of allowed system calls. In contrast, our results demonstrate that TIMELOOPS is able to tailor the system call list to specific inputs, and the degree of difference shows the typical reduction in attack surface.

When comparing system call policies, it is easiest to compare numbers of system calls in each policy, as in the analysis above. This provides a good approximation of the attack surface available to a malicious actor, but attackers do not need to access to every system call in order to be successful; they frequently require only a specific subset of system calls that is unique to the post-exploitation behavior that they desire. This
is difficult to characterize, but in attempt to do so, we pro-
vide deeper analysis of the policies generated for two of the
programs that we evaluated. We selected Nginx and the Com-
posePost service from the Social Network DeathStarBench
suite.

For both the Nginx and ComposePost programs, Timeloops
created a smaller system call policy then sysfilter. In terms of
individual system calls, our Nginx web application was allowed
to execute 40 system calls by sysfilter that were not allowed by
Timeloops. ComposePost was allowed to execute 37 system
calls by sysfilter that Timeloops did not allow. Many of these
40 and 37 system calls were benign, like exit. Others, how-
ever, may provide opportunities to compromise the con-
tainer isolation from the host system, with numerous
previous CVEs reported with associated system calls in the
Linux kernel. For example, sysfilter allowed both Nginx
and ComposePost to have access to mremap (associated
with CVE-2020-10757), sendmmsg (CVE-2011-4594), and
ftruncate (CVE-2018-18281) while Timeloops did not.
Sysfilter also allowed Nginx to execute shared
memory operations shmam and shmem (CVE-2017-5669),
and allowed ComposePost to execute clock_settime,
while Timeloops did not allow these. clock_settime not
allowed by the default Podman container filter due to its
effects on the host system settings outside of the container.
Note that the presence of previous CVEs associated with
a system call is not enough to definitively say that a system
call is or is not dangerous when executed inside a container
by an attacker; the CVEs simply show that there has been some
previous risk with allowing unrestricted access to the system
call but do not mean that the system call is still a risk, nor
that a system call with no associated CVE is not a risk. We
provide a full table of system calls allowed by each system,
along with a best-effort attempt to map the system calls to
associated Linux kernel CVEs, in Appendix A.

Some system calls are allowed by Timeloops that are
not allowed by sysfilter, in part due to the hardening
approaches applied to the program to create the oracle
services. For example, Timeloops policies include system calls
introduced by ASan, while sysfilter policies were gen-
erated from a non-ASan version of the program. The Ng-
inx Timeloops policy contained seven system calls that
were not present in the sysfilter policy, and the Compose-
Post Timeloops policy contained 13 system calls that were
not present in the sysfilter. This is contrasted to 37 and
40, respectively, that were included by sysfilter and not
Timeloops. Some of these, like open (CVE-2020-8428 [1])
and pipe (CVE-2015-1805) do have history of being abused
in prior vulnerabilities, but overall this remains a smaller at-
tacker surface.

In order to determine which system calls were specifically
introduced by the Timeloops system, we compare the Nginx
and ComposePost policies generated by Timeloops to the
system calls observed when executing the target program in a
container outside of the Timeloops system while profiling
all system calls. The Timeloops policies contain a superset
of system calls compared to the system calls observed in
the baseline profile, meaning that all system calls executed
by the program outside of Timeloops were also in the
Timeloops policies. The Timeloops policies included six
additional system calls in the Nginx policy (clock_gettime,
killed, madvise, open, readlink, sigaltstack) and seven
additional system calls in the ComposePost policy (getpid,
gettid, readlink, sched_getaffinity, sched_yield, setrlimit, sigaltstack).

We conclude that Timeloops is effective at generating
tight system call policies when compared to current state-of
the-art static analysis approaches. Timeloops does introduce
some new system calls, but at a low incidence rate.

7 Related Work

7.1 Static System Call Filtering

Static system call filtering approaches attempt to determine a
correct system call policy without executing the program,
and then apply the policy by some enforcement mecha-
nism. Sysfilter [19] is a binary analysis-based framework
that synthesizes syscall filters and applies them to applica-
tions. Like our approach, sysfilter determines developer-
intended program execution and enforces only that behavior
via seccomp-BPF filters. However, sysfilter determines
this behavior statically and analyzes call graphs to generate
their system call filters. While the authors use many tech-
niques to prune this call graph to avoid adding unreachable
system calls to their filter, it still may be too large. Abhya [36],
like sysfilter, uses static code analysis to generate system
call policies, but analyzes program source code, rather than
compiled binaries, and targets both seccomp-BPF and Pledge
policies for enforcement.

Temporal syscall filtering [24] relies on static analysis
to create syscall filters for applications. It differs from
sysfilter because it makes the observation that many sys-
tem calls that are used for the initialization of an application
are no longer required during the application’s steady state.
The authors create two filters: one filter includes all of the
initialization system calls and is applied during the initia-
lization phase of execution, and the other is applied after the
initialization concludes.

Saphire [13] presents an approach for creating system call
policies for interpreted languages. Specifically, Saphire cre-
ates sandboxes for PHP applications by analyzing PHP source
code rather than attempting to analyze the entire interpreter.
7.2 Dynamic System Call Filtering

Dynamic syscall filtering tools execute the target program with various inputs to observe which system calls are required by an application. Like TIMELOOPS, dynamic approaches execute the program to derive policies based on observations of program behavior. However, they tend to execute in distinct phases of training and deployment, while TIMELOOPS conducts its learning while able to service production workloads. ZenIDS [25] is able to learn system call policies for PHP applications with an online training period to then monitor for anomalies. Systrace [38] also uses tracing to learn policies, but enforces the policies by implementing a userspace daemon.  

7.3 Cloud and Container Security

As cloud and container environments have proliferated, new approaches have been proposed to secure them. Confine [23] is a static system call filtering technique specifically for analyzing the contents of containers to determine allowed system calls exposed to the container. Dynamic approaches have also been applied to containers [48] to profile the syscalls executed by the container during execution of automated test inputs, and to enforce a policy restricted by the profile. Work has been done to analyze and categorize [32] privilege escalation container exploits that break the resource isolation provided by container abstractions. AUTOARMOR [30] is a proposal to automatically generate policies for securing microservices by analyzing the interactions between them.

8 Future Work

8.1 Optimizations

8.1.1 Performance Optimizations

In our evaluation, we saw that most of the additional overheads that timeloops applications have in comparison to the unhardened services come from processing the first few requests. The rest of the execution after this “warmup phase” is similar to that of the unhardened services. To avoid incurring this warmup cost, we can save filters and reuse them in the future if we are confident of their correctness and can validate that they have not been compromised between uses.

Additionally, we see that it is quite costly to stop and restart containers, but this step is necessary since Seccomp filters are immutable once installed. This is a feature of Seccomp since we do not want attackers to be able to modify the Seccomp filters that are installed on a running program. This could lead to uninstalling the Seccomp filter entirely or adding system calls necessary for an attack. If we had a custom eBPF program that allowed our trusted timeloops infrastructure to modify the system call filter, then we could avoid incurring this cost every time we encounter a new system call.

8.1.2 Security Optimizations

Our current implementation of TIMELOOPS involves iteratively writing a security policy by creating an allow-list of behaviors. However, over time, the intended behavior of an application may change depending on the phase of execution, the set of inputs, the time of day or some other factor. When application behavior changes, we do not want to include old behaviors in our security policy any longer. To further tighten our security guarantees, we could implement periodic policy forgetting. This would allow us to ensure that our policy enforces tight bounds and only encapsulates current behavior.

Another assumption we make in our timeloops design is that malicious behavior is always introduced via a vulnerability that our oracle can detect. This is not always the case. Future instantiations of TIMELOOPS could use different or even multiple hardening techniques for the oracle.

Another optimization for our system call learning case study is to keep track of system call arguments. This will enforce a finer-grained security policy. Even though common system calls are used during attacks, they are often invoked with different arguments than the ones used in normal program execution.

8.2 Hardware Support

Introducing hardware support in our TIMELOOPS design may allow us to incorporate some of these optimizations with ease. For example, hardware checkpointing techniques [28] offer the ability to simply pause execution in one instance of a service and resume execution later. Instead of resending a request to the production service again, we can simply resume execution. Hardware checkpointing in conjunction with hardware support for replaying execution could save us the overheads of stopping services each time we need to consult the oracle. This support could potentially allow us to imple-
ment timeloops on services that are not stateless as well.

9 Conclusion

Creating security policies for a system is a difficult and tedious task. It requires a thorough understanding of each component of the system and how it interacts with all other components in all scenarios. This paper takes the first steps towards automatically crafting security policies for containerized microservices. We aim to learn policies for system call filtering, a technology that is becoming increasingly necessary for cloud services.

To learn policies automatically we introduced the idea of timeloopming. The idea is to: (i) re-execute a program among two variants of an application, one hardened for security, and another optimized for performance; (ii) learn execution properties from the hardened version; and (iii) use the results of the previous step to craft policies that can be enforced on the performance optimized version. Our solution takes a pragmatic stance in the trade-off between security and performance.

We demonstrated the merit and applicability of our TIMELOOPS learning scheme in three significant ways: (1) we showed that the system call allow list created by TIMELOOPS for our applications are significantly better than statically- and dynamically-generated policies, (2) we showed that the idea is compatible with, and well-suited to modern software engineering practices such the use of containerized microservices crafted using popular interpreted languages. These use cases have been the Achilles’ heel of several security studies in our community, and (3) we showed that the idea can be easily implemented by combining existing technologies.

In our community there has been significant amount of work on hardening applications. As is to be expected these hardening services come with performance and energy overheads. In an ideal world, these overheads should not matter. In reality, however, users care about performance, forcing them to make a hard choice between performance and security. In our TIMELOOPS system we use the oracle version as needed to learn security properties, and then enforce these security properties on the production version using lightweight enforcement techniques. Thus TIMELOOPS obviates the need to make this performance/security choice. With time, we expect properties and policies beyond system call filtering to use our TIMELOOPS technique.

10 Acknowledgements and Disclosures

This work was partially supported by N00014-20-1-2746, a NSF Graduate Research Fellowship, and a gift from Bloomberg. Any opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US government or commercial entities. Simha Sethumadhavan has a significant financial interest in Chip Scan Inc. Patent Pending.

References

[1] Cve-2020-8428. National Vulnerability Database.
[2] Linux sandboxing. Chromium Git Repository.
[3] Pod manager tool (podman). podman website.
[4] Pymongo 4.0.1 documentation.
[5] Systemd(1). Linux manual page.
[6] Using oauth 2.0 to access googleapis.
[7] Welcome to flask[1].
[8] What is amazon ec2? AWS User Guide for Linux Instances.
[9] Decomposing twitter: Adventures in service-oriented architecture, Jul 2013.
[10] Sceccomp security profiles for docker, Dec 2021.
[11] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. {SHARD}: Fine-grained kernel specialization with context-aware hardening. In 30th USENIX Security Symposium (USENIX Security 21), 2021.
[12] Bob Beck. Pledge, and unveil, in openbsd.
[13] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing {PHP} applications with tailored system call allowlists. In 30th USENIX Security Symposium (USENIX Security 21), 2021.
[14] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In 2019 IEEE Symposium on Security and Privacy (SP), pages 985–999. IEEE, 2019.
[15] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. In Proceedings of the 2021 on Cloud Computing Security Workshop, pages 139–151, 2021.
[16] Capsule8 Labs. An exercise in practical container escapology.
[17] Adrian Cockcroft. Evolution of microservices - craft conference, Apr 2016.
[18] Jeffrey Dean and Luiz André Barroso. The tail at scale. Commun. ACM, 56(2):74–80, feb 2013.
[20] Docker Documentation. Seccomp security profiles for docker.

[21] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 181–195, 2018.

[22] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rath, Nayan Katarkari, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud &; edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benmeur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020), pages 443–458, 2020.

[24] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In 29th USENIX Security Symposium (USENIX Security 20), pages 1749–1766. USENIX Association, August 2020.

[25] Byron Hawkins and Brian Demsky. Zenids: Introspective intrusion detection for php applications. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 232–243, 2017.

[26] Matt Hedges and Joseph Keating. Deploying python flask microservices to aws using open source tools, Apr 2021.

[27] Sungjin Kim, Byung Joon Kim, and Dong Hoon Lee. Prof-gen: Practical study on system call whitelist generation for container attack surface reduction. In 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pages 278–287. IEEE, 2021.

[28] Dirk Koch, Christian Haubelt, and Jürgen Teich. Efficient hardware checkpointing: Concepts, overhead analysis, and implementation. In Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays, FPGA ’07, page 188–196, New York, NY, USA, 2007. Association for Computing Machinery.

[29] Per Larsen, Andrei Homescu, Stefan Brunthaleral, and Michael Franz. Sok: Automated software diversity. In 2014 IEEE Symposium on Security and Privacy, pages 276–291. IEEE, 2014.

[30] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. Automatic policy generation for inter-service access control of microservices. In 30th USENIX Security Symposium (USENIX Security 21), 2021.

[31] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In 2017 USENIX Annual Technical Conference (USENIX ATC ’17), pages 1–13, 2017.

[32] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC ’18, page 418–429, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Linux Programmer’s Manual. Cgroups(7).

[34] Linux Programmer’s Manual. Seccomp(2).

[35] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In USENIX winter, volume 46, 1993.

[36] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated policy synthesis for system call sandboxing. Proc. ACM Program. Lang., 4(OOPSLA), November 2020.

[37] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. ACM SIGOPS Operating Systems Review, 27(2):72–76, 1993.

[38] Niels Provos. Improving host security with system call policies. In Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12, SSYM’03, page 18, USA, 2003. USENIX Association.

[39] Liz Rice. Container security: fundamental technology concepts that protect containerized applications. OR-eilly Media, 2020.
A System Call Policy Comparison Table

As part of our evaluation of TIMELOOPS, we generated policies for multiple programs, as described in Section 6.3. In this appendix, we list all system calls contained in the generated policies produced by TIMELOOPS and Sysfilter for our Nginx web application and the ComposePost microservice of the Social Network service benchmark. We additionally compare these policies to system calls observed when executing the programs in a container without any system call filtering, as well to the default Podman system call filter that runs. We additionally list the most recent Linux kernel CVE that we were able to identify that could be associated with the system call.

[40] Valentin Rothberg. Generate seccomp profiles for containers using podman and ebpf, Oct 2019.

[41] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. IEEE, 63(9):1278–1308, 1975.

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In 2012 {USENIX} Annual Technical Conference ({USENIX} {ATC} 12), pages 309–318, 2012.

[43] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: sanitizing for security. In 2019 IEEE Symposium on Security and Privacy (SP), pages 1275–1295. IEEE, 2019.

[44] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In 2013 IEEE Symposium on Security and Privacy, pages 48–62. IEEE, 2013.

[45] The Linux Kernel. Seccomp bpf (secure computing with filters).

[46] W3 Techs Web Technology Surveys. Usage statistics of nginx.

[47] W3 Techs Web Technology Surveys. Usage statistics of php for websites.

[48] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining sandboxes for linux containers. In 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pages 92–102, 2017.

[49] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. Mptee: Bringing flexible and efficient memory protection to intel sgx. In Proceedings of the Fifteenth European Conference on Computer Systems, pages 1–15, 2020.
Table 1: Comparison of system calls allowed by various filtering policies. If a system call does not appear in this table, it was not contained in a policy generated by TIMELOOPS, Sysfilter, or observed when the program executed. The CVE column lists one CVE, if any could be found, in the Linux kernel where the associated system call was used to trigger the vulnerability.

| syscall    | CVE          | Nginx | ComposePost |
|------------|--------------|-------|-------------|
|            | Baseline | Timeloops | Sysfilter | Baseline | Timeloops | Sysfilter | Podman |
| accept     | CVE-2017-8890 | x | x | x | x | x | x | x |
| accept4    |           | x | x | x | x | x | x | x |
| access     |           | x | x | x | x | x | x | x |
| alarm      |           | x | x | x | x | x | x | x |
| arch_prctl |           | x | x | x | x | x | x | x |
| bind       | CVE-2016-10200 | x | x | x | x | x | x | x |
| brk        | CVE-2020-9391 | x | x | x | x | x | x | x |
| capset     |           | x | x | x | x | x | x | x |
| chdir      |           | x | x | x | x | x | x | x |
| chmod      | CVE-2016-7097 | x | x | x | x | x | x | x |
| chown      | CVE-2015-3339 | x | x | x | x | x | x | x |
| clock_getres |           | x | x | x | x | x | x | x |
| clock_gettime | CVE-2011-3209 | x | x | x | x | x | x | x |
| clock_nanosleep | CVE-2009-2767 | x | x | x | x | x | x | x |
| clock_settime |           | x | x | x | x | x | x | x |
| clone      | CVE-2019-11815 | x | x | x | x | x | x | x |
| close      |           | x | x | x | x | x | x | x |
| connect    |           | x | x | x | x | x | x | x |
| dup        | CVE-2016-3750 | x | x | x | x | x | x | x |
| dup2       |           | x | x | x | x | x | x | x |
| epoll_create | CVE-2011-1083 | x | x | x | x | x | x | x |
| epoll_ctl  | CVE-2013-7446 | x | x | x | x | x | x | x |
| epoll_wait |           | x | x | x | x | x | x | x |
| eventfd2   |           | x | x | x | x | x | x | x |
| execve     | CVE-2018-14634 | x | x | x | x | x | x | x |
| exit       |           | x | x | x | x | x | x | x |
| exit_group |           | x | x | x | x | x | x | x |
| fadvise64  |           | x | x | x | x | x | x | x |
| fcntl      | CVE-2016-7118 | x | x | x | x | x | x | x |
| fstat      |           | x | x | x | x | x | x | x |
| ftruncate  | CVE-2018-18281 | x | x | x | x | x | x | x |
| futex      | CVE-2020-14381 | x | x | x | x | x | x | x |
| getcwd     |           | x | x | x | x | x | x | x |
| getdents   | CVE-2011-1593 | x | x | x | x | x | x | x |
| getdents64 |           | x | x | x | x | x | x | x |
| getegid    |           | x | x | x | x | x | x | x |
| geteuid    |           | x | x | x | x | x | x | x |
| getgid     |           | x | x | x | x | x | x | x |
| getpeername|           | x | x | x | x | x | x | x |
| getpid     |           | x | x | x | x | x | x | x |
| getppid    |           | x | x | x | x | x | x | x |
| getrandom  |           | x | x | x | x | x | x | x |
| getrlimit  |           | x | x | x | x | x | x | x |
| getsockname| CVE-2021-38208 | x | x | x | x | x | x | x |
| getssockopt | CVE-2021-20194 | x | x | x | x | x | x | x |
| gettid     |           | x | x | x | x | x | x | x |
| gettimeofday|           | x | x | x | x | x | x | x |
Table 1: Comparison of system calls allowed by various filtering policies. If a system call does not appear in this table, it was not contained in a policy generated by TIMELOOPS, Sysfilter, or observed when the program executed. The CVE column lists one CVE, if any could be found, in the Linux kernel where the associated system call was used to trigger the vulnerability.

| syscall       | CVE                      | Nginx | Baseline | Timeloops | Sysfilter | Podman | ComposePost | Baseline | Timeloops | Sysfilter | Podman |
|---------------|--------------------------|-------|----------|-----------|-----------|--------|-------------|----------|-----------|-----------|--------|
| getuid        |                          | x     | x        | x         |           |        |             |          |           |           |        |
| ioctl         | numerous drivers         | x     | x        | x         |           |        |             |          |           |           |        |
| kill          |                          | x     | x        | x         |           |        |             |          |           |           |        |
| listen        |                          | x     | x        | x         | x         | x      |             |          |           |           |        |
| lseek         | CVE-2013-3301            | x     | x        | x         |           |        |             |          |           |           |        |
| lstat         |                          | x     | x        | x         |           |        |             |          |           |           |        |
| madvise       | CVE-2016-5195            | x     | x        | x         | x         | x      |             |          |           |           |        |
| mkdir         |                          | x     | x        |           |           |        |             |          |           |           |        |
| mmap          | CVE-2018-7740            | x     | x        | x         | x         | x      |             |          |           |           |        |
| mprotect      | CVE-2010-4169            | x     | x        | x         | x         | x      |             |          |           |           |        |
| mremap        | CVE-2020-10757           | x     |           |           |           |        |             |          |           |           |        |
| munmap        | CVE-2020-29369           | x     | x        | x         | x         | x      |             |          |           |           |        |
| nanosleep     |                          |       |          |           |           |        |             |          |           |           | x      |
| newfstatat    |                          |       |           |           |           |        |             |          |           |           | x      |
| open          | CVE-2020-8428            | x     | x        | x         |           |        |             |          |           |           | x      |
| openat        | CVE-2020-10768           | x     | x        | x         |           |        |             |          |           |           | x      |
| pause         |                          |       |           |           |           |        |             |          |           |           | x      |
| pipe          | CVE-2015-1805            | x     | x        |           |           |        |             |          |           |           | x      |
| poll          |                          | x     | x        | x         | x         | x      |             |          |           |           | x      |
| prctl         | CVE-2020-10768           | x     | x        | x         |           |        |             |          |           |           | x      |
| pread64       |                          | x     | x        |           |           |        |             |          |           |           | x      |
| prlimit64     |                          | x     | x        |           |           |        |             |          |           |           | x      |
| pselect6      |                          | x     | x        |           |           |        |             |          |           |           | x      |
| pwrite64      |                          | x     | x        |           |           |        |             |          |           |           | x      |
| pwritev       |                          |       |           |           |           |        |             |          |           |           | x      |
| read          |                          | x     | x        | x         | x         | x      |             |          |           |           | x      |
| readlink      | CVE-2011-4077            | x     |           |           |           |        |             |          |           |           | x      |
| readv         | CVE-2008-3535            | x     | x        |           |           |        |             |          |           |           | x      |
| recvfrom      | CVE-2013-1979            | x     | x        | x         | x         | x      |             |          |           |           | x      |
| recvmsg       | CVE-2013-1979            | x     |           |           |           |        |             |          |           |           | x      |
| rename        | CVE-2016-6198            | x     |           |           |           |        |             |          |           |           | x      |
| restart_syscall| CVE-2014-3180           | x     |           |           |           |        |             |          |           |           | x      |
| rmdir         |                          |       |           |           |           |        |             |          |           |           | x      |
| rt_siganction |                          | x     | x        | x         | x         | x      |             |          |           |           | x      |
| rt_sigsuspend | CVE-2017-15537           | x     |           |           |           |        |             |          |           |           | x      |
| sched_getparam|                          | x     | x        |           |           |        |             |          |           |           | x      |
| sched_setaffinity |              | x     |           |           |           |        |             |          |           |           | x      |
| sched_yield | CVE-2021-26708           | x     |           |           |           |        |             |          |           |           | x      |
Table 1: Comparison of system calls allowed by various filtering policies. If a system call does not appear in this table, it was not contained in a policy generated by Timeloops, Sysfilter, or observed when the program executed. The CVE column lists one CVE, if any could be found, in the Linux kernel where the associated system call was used to trigger the vulnerability.

| syscall            | CVE       | Nginx Baseline | Timeloops | Sysfilter | ComposePost Baseline | Timeloops | Sysfilter | Podman |
|--------------------|-----------|----------------|-----------|-----------|----------------------|-----------|-----------|--------|
| sendmmsg           | CVE-2011-4594 | x              |           |           |                      |           |           |        |
| sendmsg            | CVE-2017-17712 | x              |           |           |                      |           |           |        |
| sendto             | CVE-2017-17712 | x              | x         |           |                      |           |           |        |
| set_robust_list    |            | x              | x         |           |                      | x         |           |        |
| set_tid_address    |            | x              | x         |           |                      | x         |           |        |
| setgid             | CVE-2021-32760 | x              | x         | x         |                      | x         |           |        |
| setgroups          | CVE-2018-7169 | x              | x         |           |                      |           |           |        |
| setitimer          |            | x              | x         |           |                      |           |           |        |
| setpriority        |            |               |           |           |                      |           |           |        |
| setregid           |            | x              | x         |           |                      |           |           |        |
| setresgid          |CVE-2019-18684| x              | x         |           |                      | x         |           |        |
| setresuid          | CVE-2011-3145 | x              |           |           |                      |           |           |        |
| setrlimit          |            |               |           |           |                      |           |           |        |
| setsid             | CVE-2005-0178 | x              | x         | x         |                      |           |           |        |
| setsockopt         | CVE-2016-4998 | x              | x         |           |                      | x         |           |        |
| setuid             | CVE-2013-6825 | x              | x         | x         |                      |           | x         |        |
| shmat              | CVE-2017-5669 | x              |           |           |                      |           |           |        |
| shmdt              |            |               |           |           |                      |           |           |        |
| shmget             | CVE-2017-5669 | x              |           |           |                      |           |           |        |
| shutdown           |            | x              | x         | x         |                      | x         |           |        |
| sigaltstack        | CVE-2009-2847 | x              |           | x         |                      |           | x         |        |
| socket             | CVE-2017-9074 | x              | x         | x         |                      |           | x         |        |
| socketpair         | CVE-2010-4249 | x              | x         | x         |                      |           |           |        |
| stat               |            | x              | x         | x         |                      |           | x         |        |
| statfs             |            |               |           |           |                      |           |           |        |
| sysfsinfo          |            | x              | x         | x         |                      |           |           |        |
| tgkill             | CVE-2013-2141 | x              |           | x         |                      |           |           |        |
| time               |            |               |           |           |                      |           |           |        |
| times              |            | x              | x         |           |                      |           |           |        |
| umask              | CVE-2020-35513 | x              |           |           |                      |           |           |        |
| uname              | CVE-2012-0957 | x              | x         | x         |                      | x         |           |        |
| unlink             | CVE-2016-6197 | x              |           |           |                      |           | x         |        |
| utimes             |            |               |           |           |                      |           |           |        |
| vfork              | CVE-2005-3106 | x              |           | x         |                      |           |           |        |
| wait4              |            | x              | x         | x         |                      |           |           |        |
| write              |            | x              | x         | x         |                      | x         |           |        |
| writenv            | CVE-2016-9755 | x              | x         |           |                      |           |           |        |