SYGMA: Analysis-Guided Preemptive Enforcement of Stateful Syscall Policies in Containers

Jin Her*
Incheon National University
Incheon, Republic of Korea
gjwls0787@inu.ac.kr

Chihyeon Cho*
Incheon National University
Incheon, Republic of Korea
gsm07231@inu.ac.kr

Jaehyun Nam†
Dankook University
Yongin-si, Republic of Korea
namjh@dankook.ac.kr

Seungsoo Lee†
Incheon National University
Incheon, Republic of Korea
seungsoo@inu.ac.kr

Abstract

Container technologies have become fundamental to modern cloud applications, but their reliance on a shared host kernel exposes workloads to security threats through the syscall interface. Although syscall filtering mechanisms such as Seccomp-BPF reduce the attack surface by restricting individual syscalls, they cannot maintain execution-history state or natively encode state-dependent syscall ordering. Existing stateful syscall filtering approaches further face three challenges in container environments: incomplete preemptive syscall enforcement, limited stateful protection during container initialization, and inconsistent synchronization between policies, runtime states, and container lifecycle events. This work presents SYGMA, a stateful syscall enforcement framework that combines analysis-guided hybrid enforcement, stateful initialization policies, and lifecycle-synchronized policy-state management to provide continuous protection throughout the container lifecycle. Our evaluation shows that SYGMA provides preemptive enforcement across the evaluated syscall space, reduces the number of syscalls permitted during initialization by up to 67% while keeping the false rejection rate at most 6.0%, eliminates the startup policy activation gap, and incurs only 1.8–5.1% runtime performance overhead across four representative server applications.

CCS Concepts

• **Security and privacy → Operating systems security**.

Keywords

cloud-native security; system call filtering

1 Introduction

The widespread adoption of microservice architectures and advances in orchestration platforms such as Kubernetes [67] have accelerated the efficient deployment and scaling of modern cloud applications [40]. In this environment, containers have become core execution units for cloud-native applications by providing lightweight isolation and high portability. However, because containers share the host kernel and syscall interface [12, 25, 61, 63], an attacker who exploits kernel vulnerabilities or abuses syscalls can cross the container isolation boundary and affect the host and the broader system. Attackers have exploited these properties to mount various attacks, including container escapes [1, 26, 43, 44, 55]. To mitigate these threats, recent studies have proposed various techniques that restrict container behavior [37, 57, 64].

Among these defenses, syscall [1] filtering is a primary technique for reducing the attack surface by permitting only the syscalls required by an application. Seccomp-BPF [57], a widely used syscall filtering mechanism in container environments, blocks syscalls unnecessary for normal execution according to policy-defined rules. However, Seccomp-BPF evaluates each syscall independently, making it difficult to encode execution history or state-dependent syscall ordering in a policy. Consequently, Seccomp-BPF alone cannot effectively prevent attacks that invoke individually permitted syscalls in an abnormal order [4, 27, 32, 34].

To address these limitations, research on stateful syscall filtering, which jointly considers execution state and invocation order, has advanced primarily in policy generation [4, 8, 68] and runtime enforcement [4, 27, 28, 34]. However, deploying existing techniques in container environments still faces three limitations. First, depending on per-syscall kernel execution paths, existing enforcement mechanisms [24, 28, 41, 64] provide limited preemptive enforcement coverage or allow a syscall to continue executing after detecting a policy violation and initiating a blocking action. Second, existing binary-derived stateful policy generation techniques primarily analyze execution of the main application binary and therefore do not adequately protect the syscalls and transitions invoked during container initialization before main execution [11, 21, 39]. Third, if policy activation, phase transitions, and state cleanup are not synchronized with container creation, the init-to-main transition, termination, and restart, syscalls may execute without the intended policy or be evaluated against an incorrect runtime state [28, 64].

Motivated by these limitations, we present SYGMA, an end-to-end framework for deploying and enforcing binary-derived stateful syscall policies throughout the container lifecycle. SYGMA analyzes the source code and build configuration of the target kernel to classify per-syscall execution paths and enforcement boundaries, then routes each syscall through Syscall-Entry SIGKILL, LSM-BPF, or Ptrace-Coupled SIGKILL to preemptively block policy-violating syscalls during syscall execution before they return successfully. It also dynamically analyzes the container initialization phase to

*Both authors contributed equally to this research. / †Corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASIA CCS '27, Macau
© 2027 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

[1] In this paper, we use the terms system call and syscall interchangeably.

generate a separate stateful initialization policy, providing continuous stateful protection from initialization through main execution while preserving the semantics of the existing main-execution policy. Finally, SYGMA activates, switches, and removes policies and runtime state in synchronization with container lifecycle events to provide consistent policy enforcement throughout the lifecycle.

We evaluate SYGMA in terms of preemptive enforcement coverage, policy precision, startup policy activation, and runtime performance. The results show that SYGMA provides preemptive enforcement for every syscall supported by the kernel, reduces the syscall attack surface during initialization by up to 67% over existing approaches, and maintains a false rejection rate of up to 6.0% during normal container deployment. It also eliminates the policy activation gap at container startup, ensuring that no application syscall executes before policy activation. Across four server applications, SYGMA incurs 1.8–5.1% runtime overhead, which is lower than that of the enforcement baseline. These results demonstrate that SYGMA provides comprehensive stateful syscall enforcement with lifecycle-synchronized policy management while maintaining practical runtime efficiency.

The key contributions of this work are as follows:

- We systematically analyze the per-syscall execution paths and enforcement boundaries of LSM-BPF and Conventional SIGKILL, and design an analysis-guided scheme that routes each syscall to Syscall-Entry SIGKILL, LSM-BPF, or Ptrace-Coupled SIGKILL.
- We design and implement SYGMA, which uses the resulting per-syscall classification to achieve preemptive enforcement across all syscalls supported by the target kernel, extends stateful protection to container initialization, and synchronizes policies and runtime state with container lifecycle.
- We evaluate SYGMA on widely adopted containerized server applications and microservice workloads. SYGMA provides preemptive enforcement for all syscalls supported by the target kernel, reduces the initialization-phase syscall attack surface by up to 67%, eliminates the policy activation gap, and incurs 1.8–5.1% runtime overhead.

2 Background

2.1 Container Execution Phases and Syscall Behavior

A microservice architecture (MSA) [40] decomposes an application into multiple services that can be independently deployed and executed, with each service typically deployed in a separate container. Containers use Linux kernel mechanisms such as namespaces [66] and cgroups [65] to provide isolated execution environments for individual services, but they share the host kernel and syscall interface. In a typical container with a separate PID namespace, the process launched by the entrypoint runs as PID 1 and prepares the service environment through a shell script, init process, or setup utility. When the main binary is subsequently executed through `execve()`, container-level initialization ends and the application's main execution begins. Accordingly, we divide container execution into an *initialization phase* and a *main execution phase*, using the execution of the main binary as the phase boundary.

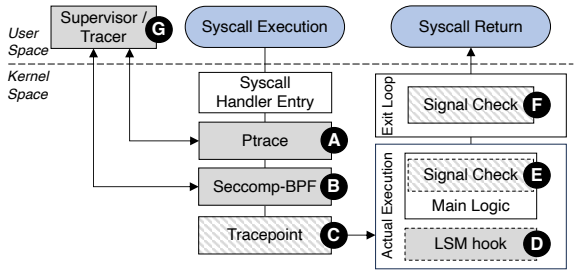


Figure 1: Overview of the syscall execution path from invocation to kernel processing. Actual Execution represents the kernel's main syscall logic, while Exit Loop represents the return path where signal checks are performed.

During the initialization phase, the container prepares the service environment before executing the main binary. This process generates configurations, sets directory permissions and environment variables, and prepares auxiliary processes and threads. These tasks invoke file-, memory-, privilege-, and process-related syscalls, many of which are specific to initialization [21, 39]. In contrast, during the main execution phase, the microservice application continuously performs its primary functions, including network communication, request processing, and data computation. The two phases therefore differ not only in their execution purposes but also in the types, frequencies, and ordering of the syscalls they require.

2.2 Evolution of Syscall Policy Generation

Prior work uses static and dynamic analyses of container workloads to identify syscalls required for normal execution and automatically derive allowlist policies [6, 9, 15, 21, 49, 70, 71]. Subsequent studies distinguish container execution phases and generate more precise phase-specific allowlists from the syscalls invoked in each phase [22, 23, 39, 52]. These allowlists are typically converted into Seccomp-BPF [57] profiles, whose default action and per-syscall rules block syscalls unnecessary for normal execution and reduce the container's attack surface. However, such set-membership policies cannot represent the current execution state or syscall ordering. To address this limitation, more recent work incorporates execution states and syscall sequences observed during normal execution [4, 8, 60, 68]. These approaches combine static analysis of application binaries with runtime information to model normal execution flows and derive stateful policies. Some construct a Deterministic Finite Automaton (DFA) [58] that encodes execution states, the syscalls permitted in each state, and transitions between states [4, 60, 68], whereas others represent syscall precedence as a graph that specifies the permitted successors of each syscall [8]. Binary-derived stateful policies therefore capture both syscall permissions and execution-state-dependent ordering constraints.

2.3 Programmable Syscall Policy Enforcement

Unlike the stateless allowlists described above, enforcing stateful policies such as DFAs and phase-specific policies requires a programmable enforcement mechanism that maintains runtime state and evaluates each syscall and its corresponding state transition.

Table 1: Comparative analysis of security properties across syscall enforcement mechanisms.

Policy Enforcement Property	User-Space-Assisted		eBPF-Based	
	Seccomp Notify	Ptrace	Conventional SIGKILL	LSM-BPF
Kernel-Resident TCB	✗	✗	✓	✓
Low-latency Decision-Making	✗	✗	✓	✓
Complete Syscall Coverage	✓	✓	✓	✗
Preemptive Syscall Enforcement	✓	✓	✗	✗

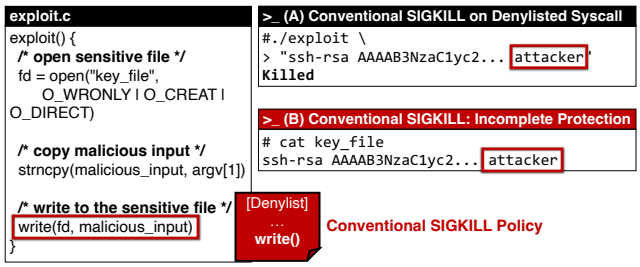


Figure 2: Limitation of Conventional SIGKILL. Despite terminating the process (A), the attacker successfully modifies the target file before termination (B).

Linux mechanisms applicable to these policies can be broadly categorized as user-space-assisted enforcement and extended Berkeley Packet Filter (eBPF)-based enforcement. Figure 1 shows where ptrace, Seccomp-BPF, tracepoints, and Linux Security Module (LSM) hooks operate along the syscall execution path, as well as the control flows for user-space mediation and signal-based termination.

User-Space-Assisted Enforcement. Ptrace and Seccomp Notify suspend syscall execution and enable programmable policy decisions in user space [5, 42]. At syscall entry, Ptrace places the target process in a system-call-stop state at point A, allowing the user-space tracer at point G to inspect the syscall number, arguments, and execution context before resuming execution. The tracer can modify the syscall number or related state stored in registers to skip the target syscall or replace it with another syscall. When a Seccomp-BPF filter returns a user notification at point B, Seccomp Notify suspends the calling thread and delegates the policy decision to the user-space supervisor at point G. After inspecting the syscall information, the supervisor returns a verdict to the kernel, such as permitting the syscall, returning an error, or supplying a specified return value. The kernel then controls the subsequent execution of the suspended syscall according to this verdict.

eBPF-Based Enforcement. Several syscall enforcement mechanisms use LSM-BPF or tracing-based eBPF programs [17, 28, 41, 64]. LSM-BPF attaches eBPF programs to LSM hooks at which security-sensitive operations on kernel objects such as files, processes, and sockets occur. When syscall execution reaches such a hook, the program evaluates the policy and, upon detecting a violation, returns a negative error value to deny the corresponding operation (D in Figure 1) [28]. In contrast, tracing-based mechanisms attach eBPF programs to tracepoints or kprobes to observe syscall events and request process termination by sending SIGKILL upon detecting a policy violation [41, 64]. Specifically, when a violation is detected at point C, the termination signal is marked as pending. The signal is subsequently processed by a signal check during syscall execution at point E or by a signal check in the exit loop at point F before the process returns to user space, terminating the process.

3 Problem Statements

Deploying a stateful syscall policy in a container environment requires more than policy generation. It also requires reliable enforcement of policy semantics, extension of stateful protection to container initialization, and policy-state management throughout the container lifecycle. We formulate these requirements as C1–C3 in terms of syscall enforcement, container initialization, and lifecycle-consistent policy-state management, respectively.

C1: Limitations of Existing Syscall Enforcement Mechanisms. Table 1 compares existing syscall enforcement mechanisms with respect to syscall coverage, preemptive enforcement, policy-decision placement, and decision latency. User-space-assisted mechanisms such as Ptrace and Seccomp Notify can suspend syscall execution and delegate programmable policy decisions to a user-space monitor. However, because the monitor maintains policy state and produces enforcement decisions, its correctness and integrity directly affect enforcement reliability. A faulty or compromised monitor may permit policy-violating syscalls or incorrectly block legitimate syscalls, extending the policy-decision Trusted Computing Base (TCB) beyond the kernel into user space [20, 32]. Moreover, each policy evaluation incurs control transfers between the kernel and user space, monitor scheduling, and context switching, resulting in additional decision latency and runtime overhead [32].

In contrast, LSM-BPF and Conventional SIGKILL enforcement operate in the kernel, avoiding user-space mediation and thereby reducing decision latency and runtime overhead. However, in-kernel enforcement alone does not guarantee preemptive blocking for every syscall. LSM-BPF can prevent a policy-violating syscall from completing only if every execution path traverses a veto-capable LSM hook and denial at that hook propagates to the syscall as a failure. Some syscalls have no applicable LSM hook, and even when one exists, its invocation may depend on the execution path. Conventional SIGKILL-based enforcement can observe a broad range of syscall events, but policy-violation detection and process termination occur at different times [27]. Consequently, a policy-violating syscall may complete its execution before the signal is handled, leaving persistent changes to system state despite the subsequent process termination. In the `write` example in Figure 2, the process is eventually terminated, but the sensitive file is modified beforehand.

Therefore, achieving preemptive syscall enforcement requires *systematically analyzing per-syscall kernel execution paths and the corresponding enforcement boundaries of LSM-BPF and Conventional SIGKILL* to identify coverage gaps and determine an appropriate enforcement path for each syscall.

C2: Stateful Policy Gap during Container Initialization. Prior container-aware studies analyze container images and include syscalls invoked during the initialization phase in their policies [21].

Listing 1: Example of a third-party stateful syscall policy for the main-execution phase [4]

```
"startState": "INIT",
"states": {
  "INIT": {
    "allowedSyscalls": ["brk","openat","mmap","prctl"],
    "transitions": {
      "brk": "INIT", "openat": "INIT",
      "mmap": "INIT", "prctl": "EVENT_LOOP"
    }
  },
  "EVENT_LOOP": {
    "allowedSyscalls": ["epoll_wait","accept4"],
    "transitions": {
      "epoll_wait": "EVENT_LOOP",
      "accept4": "HANDLING_CONN",
    }
  }
  ...
}
```

However, most approaches represent the analysis results as stateless allowlists and therefore cannot constrain allowed syscalls invoked in an abnormal order. To avoid omitting syscalls required for normal startup, these approaches may also combine syscalls observed across multiple execution paths, permitting more syscalls than a specific execution requires. Consequently, existing approaches cannot provide consistent stateful protection from container initialization through main execution.

C3: Lifecycle Misalignment of Active Policies and Runtime State. Stateful policy enforcement in containers requires the active policy and runtime state to remain synchronized with container lifecycle events. During container creation, asynchronous policy installation by an external control-plane component, such as a Kubernetes Operator, may allow the entrypoint and early syscalls to execute before policy activation, creating a policy activation gap [27]. Critically, during the transition from initialization to main execution, misalignment among the execution of the main binary, the corresponding policy switch, and the runtime-state transition may cause main-execution syscalls to be evaluated under an incorrect policy or runtime state, resulting in a phase mismatch. Also, after termination or restart, policies or thread-level states remaining in kernel maps become stale and may accumulate or be incorrectly associated with a new instance if identifiers are reused.

4 SYGMA Design

We propose SYGMA, an end-to-end framework for deploying and enforcing binary-derived stateful syscall policies generated by an external generator in container environments. This section first defines the threat model and policy scope of SYGMA, and then presents the design requirements derived from C1–C3 and the overall system architecture and workflow that satisfy them.

4.1 Threat Model and Policy Scope

We trust the host kernel, container runtime, orchestration layer, and all SYGMA components. SYGMA protects individual *protected containers*, each being the container instance to which SYGMA applies syscall policies and whose lifecycle it manages. An attacker may compromise a workload inside such a container and attempt to invoke a syscall that the policy disallows. Compromise of trusted components, including the host kernel and container runtime, is outside the scope of this work.

SYGMA receives the main-execution policy shown in Listing 1, generated by an external binary-analysis-based generator [4, 8, 68] and deployed by an administrator. The policy defines the syscalls allowed in each state and the state transitions triggered by their invocation. For example, in Listing 1, `prctl` transitions `INIT` to `EVENT_LOOP`, where `accept4` transitions to `HANDLING_CONN`. SYGMA treats this policy as trusted input and independently maintains runtime state for each thread. The accuracy and completeness of policy generation are outside the scope of this work.

4.2 Design Considerations

*DC1) Analysis-Guided Preemptive Syscall Enforcement (**C1**).*

It should preemptively block policy-violating syscalls during syscall execution before they return successfully. To achieve this, it should analyze per-syscall kernel execution paths and enforcement boundaries to identify where each violating syscall can be intercepted and blocked during execution. It should also perform policy decisions and runtime-state transitions in the kernel to reduce reliance on user-space monitors and minimize context-switch overhead.

*DC2) Stateful Policy Extension to Initialization (**C2**).*

It should preserve the semantics of the main-execution policy generated by the external generator while complementing it with a separate stateful policy for initialization behavior that the policy does not cover. The initialization policy should constrain not only the syscalls required for normal startup but also the transitions among them, thereby representing initialization behavior more strictly than a stateless allowlist.

*DC3) Lifecycle-Synchronized Policy-State Management (**C3**).*

It should ensure that active policies and runtime states remain synchronized with container lifecycle events. It also activates the initialization policy and initial state before execution, switches to the main-execution policy upon main binary launch, and removes all associated policies and thread states on termination or restart to prevent stale state reuse.

4.3 Overall Architecture

As shown in Figure 3, SYGMA consists of four components: the *Analyzer*, *Operator*, *Agent*, and *Filter*. The Analyzer performs a one-time offline analysis for each target kernel, while the Operator manages protected containers and their policies in the Kubernetes control plane. The Agent and Filter run in the user space and kernel of each worker node, respectively, and install and enforce policies in synchronization with the container lifecycle.

SYGMA **Analyzer.** The Analyzer examines the source code and build configuration of the target kernel to identify the per-syscall enforcement boundaries of LSM-BPF and Conventional SIGKILL, producing a syscall classification. This analysis is performed once for each target kernel build. At runtime, SYGMA uses the resulting classification to select the enforcement path for each syscall and configure the Policy Enforcer for that kernel build.

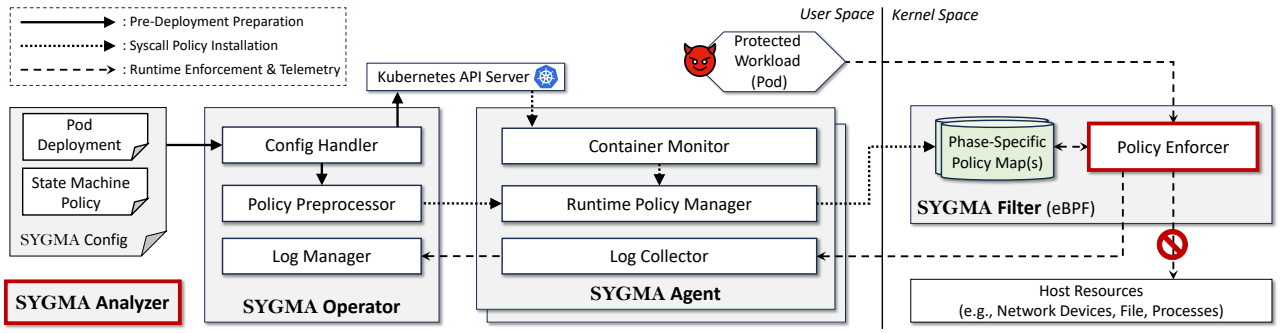


Figure 3: Overall architecture and workflow of SYGMA, which comprises the Analyzer, Operator, Agent, and Filter. Red outlines mark the Analyzer and the Policy Enforcer, where the Policy Enforcer is configured using the results of a one-time target-kernel analysis performed by the Analyzer. Policy deployment proceeds in three stages: (*i*) pre-deployment preparation, (*ii*) syscall policy installation, and (*iii*) runtime enforcement & telemetry.

SYGMA **Operator.** The Operator is a control-plane component that manages protected container configurations and syscall policies. The *Config Handler* receives a SYGMA Config consisting of a *Pod Deployment* and a *State Machine Policy* generated by an external generator, and identifies the target workload to which the config applies. Appendix A describes the structure of the SYGMA Config in detail. Before deploying the protected container, it also coordinates initialization profiling and collects information about syscalls invoked before the main binary executes to construct the initialization policy. The *Policy Preprocessor* prepares the initialization and main-execution policies in a runtime representation usable by the Filter, while the *Log Manager* collects policy violations and runtime events reported by the Agent.

SYGMA **Agent.** The Agent manages the lifecycle of protected containers and kernel-side policy state on each worker node. The *Container Monitor* tracks the deployment status of protected containers through the Kubernetes API Server and forwards pre-start and termination events from the container runtime to the Runtime Policy Manager. Before a container's entrypoint executes, the *Runtime Policy Manager* retrieves its policies from the Operator and installs them in the Filter. It subsequently updates or removes the policies and runtime state according to the container lifecycle. The *Log Collector* collects policy violations and runtime events generated by the Filter and forwards them to the Log Manager. For the Ptrace-Coupled SIGKILL path detailed in Section 6.1, the Agent also creates and supervises a trusted user-space Ptrace tracer.

SYGMA **Filter.** The Filter is an eBPF-based syscall enforcement component that runs in the kernel of each worker node and consists of *Phase-Specific Policy Map(s)* and a *Policy Enforcer*. The policy maps maintain the initialization policy, main-execution policy, and thread-level runtime state for each protected container. The Policy Enforcer evaluates syscalls against the active policy and applies the appropriate enforcement path according to the syscall classification generated by the Analyzer. It also records policy violations and runtime events required for lifecycle management, which the Agent's Log Collector subsequently collects.

Workflow. Figure 3 illustrates policy deployment for a protected container, which proceeds in the following three stages.

1) Pre-Deployment Preparation. When an administrator deploys a SYGMA Config, the Operator prepares the policies for the protected container. It first receives the main-execution policy generated by an external generator and profiles the same workload before deploying the protected container to collect initialization syscalls and transitions until the main binary executes. The Operator then generates an initialization policy from the collected information and combines it with the main-execution policy to construct the phase-specific policies for the workload.

2) Syscall Policy Installation. When a policy-enabled Pod is assigned to a worker node and its container prepares to execute, the Agent on that node detects the pre-start lifecycle event and retrieves the registered policies using the Pod and container information. The Agent associates the initialization and main-execution policies with the container instance, installs them in the Filter, and activates the initialization policy and initial state. It allows the container's entrypoint to proceed only after completing the required policy-state installation, ensuring that the policy covers the initial syscalls.

3) Runtime Enforcement & Telemetry. When a protected container invokes a syscall, the Filter determines whether the syscall and its state transition are allowed based on the active policy and thread-level runtime state. It then enforces the decision through the enforcement path corresponding to the syscall classification. Upon detecting `execve()` of the registered main binary, the Filter switches the active policy and runtime state from the initialization policy to the main-execution policy. The Agent forwards policy violations and required runtime events to the Operator and propagates administrator-approved policy updates to the Filter. When the container terminates, the Agent removes the policy and runtime state associated with that instance and applies a fresh policy and initial state to any restarted instance.

5 Per-Syscall Analysis of Preemptive Enforceability

To satisfy DC1 in Section 4.2, the SYGMA Analyzer automatically analyzes the per-syscall preemptive enforceability of LSM-BPF and Conventional SIGKILL. Because the analysis takes the source code and build configuration of the target kernel as input, it is not tied to

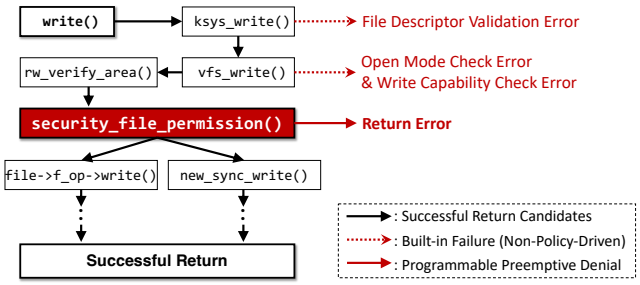


Figure 4: Simplified execution paths for the `write()` syscall and the preemptive enforcement point. Black arrows denote successful return candidates, red dotted arrows denote syscall-internal failure paths, and the red solid arrow denotes preemptive denial at `security_file_permission()`.

a specific kernel version. This section evaluates a Linux 5.15 x86-64 build and analyzes 343 of the 360 syscalls defined in its syscall table, excluding 17 unavailable syscalls.

5.1 Analysis Scope and Enforceability Criteria

Figure 4 illustrates the analysis scope using `write()`. The Analyzer incorporates the target-kernel configuration into an interprocedural control-flow analysis of each syscall implementation and defines all execution paths that can return successfully as *successful return candidates*. It classifies a syscall as preemptively enforceable by a mechanism only if the mechanism can reliably block every successful return candidate during syscall execution before the syscall returns successfully. The analysis excludes paths that fail syscall validation checks, such as those involving an invalid file descriptor or file mode, because they cannot reach a successful return.

For LSM-BPF, the Analyzer checks whether each successful return candidate invokes an LSM hook capable of denying the operation during syscall execution. The hook should support attaching a BPF program and rejecting the current operation by returning a negative error value. The Analyzer also verifies that the denial is neither ignored nor converted into success during subsequent execution and ultimately causes the syscall to fail. Because different paths may invoke different LSM hooks, all candidates need not traverse the same hook. However, the Analyzer classifies a syscall as preemptively enforceable only if an applicable LSM hook can prevent successful return along every successful return candidate. For `write()` in Figure 4, all candidates traverse `security_file_permission()` as a bottleneck, and denial at this hook causes the syscall to fail. The Analyzer therefore classifies `write()` as preemptively enforceable.

For Conventional SIGKILL, the Analyzer checks whether every successful return candidate contains a fatal-signal check during syscall execution before the syscall returns successfully(E in Figure 1). If the process has a pending fatal signal at this point, it terminates instead of allowing the syscall to return successfully, thereby providing preemptive enforcement. The Analyzer also verifies that the detected signal is not ignored during subsequent execution and leads to actual process termination. As with LSM hooks, different paths may traverse different signal-check points. However, the Analyzer classifies a syscall as preemptively enforceable with Conventional SIGKILL only if every successful return candidate satisfies these conditions. Because `write()` in Figure 4 has no such fatal-signal check point, the Analyzer classifies it as non-enforceable with Conventional SIGKILL.

Table 2: Counts of syscalls classified as *Preemptively Enforceable, Non-Enforceable,* or *Not Available* under LSM-BPF and Conventional SIGKILL in the evaluated Linux 5.15 x86-64.

Mechanism	Preemptively Enforceable	Non-Enforceable	N/A
LSM-BPF	146	197	17
Conventional SIGKILL	4	339	17

5.2 Preemptive Enforcement Coverage

Based on the mechanism-specific criteria, the Analyzer classifies each syscall as *Preemptively Enforceable*, *Non-Enforceable*, or *Not Available*. It excludes syscalls not implemented in the target kernel and classifies them as *Not Available*. Among implemented syscalls, it assigns Preemptively Enforceable only when the mechanism can block every successful return candidate during syscall execution before the syscall returns successfully; otherwise, it assigns *Non-Enforceable*. The latter category includes syscalls with paths that do not traverse an enforcement point or can still return successfully after enforcement, as well as syscalls for which static-analysis limitations, such as unresolved indirect call targets, prevent the Analyzer from establishing blocking capability. Thus, the Analyzer conservatively classifies a syscall as *Non-Enforceable* unless blocking is guaranteed across all candidates.

Table 2 shows the per-syscall enforcement results statically established by the Analyzer. Under LSM-BPF, the Analyzer classifies 146 syscalls as *Preemptively Enforceable* and the remaining 197 as *Non-Enforceable*. Under Conventional SIGKILL, it classifies only four syscalls, `clone()`, `clone3()`, `fork()`, and `vfork()`, as *Preemptively Enforceable* and the remaining 339 as *Non-Enforceable*. We refer to the enforcement of these four syscalls, in which SIGKILL is marked as pending at the syscall-entry tracepoint, as *Syscall-Entry SIGKILL*. Because the Analyzer conservatively classifies all syscalls with unestablished blocking capability as *Non-Enforceable*, these results represent a lower bound on the preemptive enforcement coverage established through static analysis. Although LSM-BPF provides broader enforcement coverage than Conventional SIGKILL, the two mechanisms alone cannot establish preemptive enforceability for all syscalls. This finding indicates the need for an additional enforcement mechanism for syscalls whose preemptive enforceability cannot be established with existing ones. Appendix B describes the implementation of the Analyzer, and the per-syscall classification results together with the evidence supporting each decision are available in the artifact described in the Open Science section.

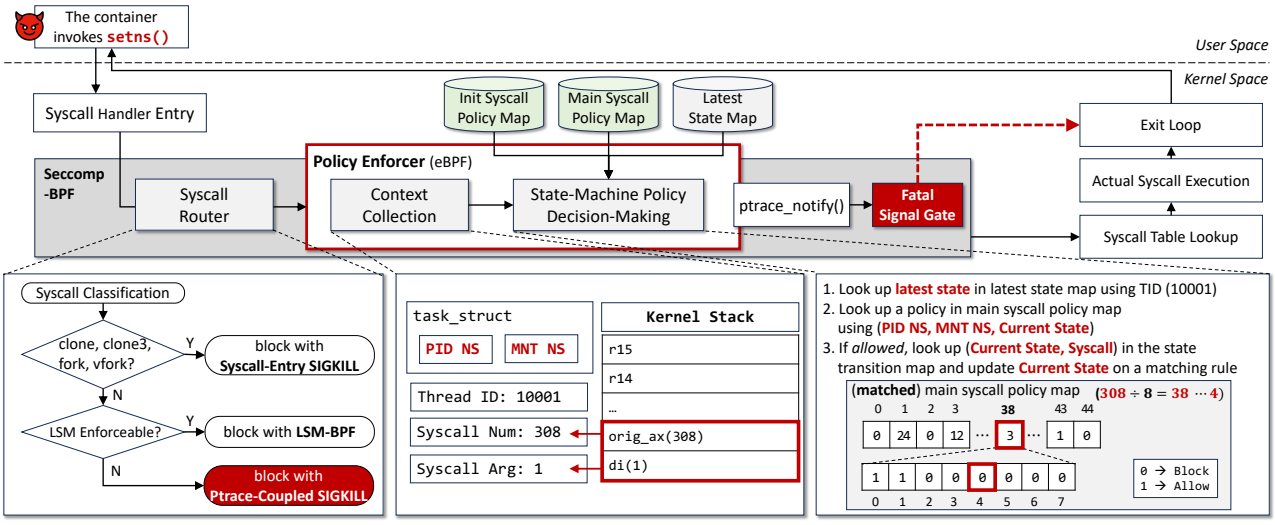


Figure 5: Example of SYGMA syscall enforcement for `setns()`. The syscall router routes each syscall to Syscall-Entry SIGKILL, LSM-BPF, or Ptrace-Coupled SIGKILL; `setns()` follows the third path. The Policy Enforcer extracts the syscall context from the register state, evaluates the state-machine policy, and, upon a DENY decision, sets a fatal signal so that the subsequent Fatal Signal Gate diverts control to the exit loop before actual syscall execution.

6 SYGMA System Details

6.1 Analysis-Guided Hybrid Syscall Enforcement

The analysis in Section 5.2 shows that LSM-BPF and Conventional SIGKILL alone cannot establish preemptive enforceability for all syscalls. Addressing this coverage gap requires an additional enforcement mechanism that blocks the remaining syscalls before actual syscall execution while retaining in-kernel policy decisions and low enforcement overhead.

Ptrace-Coupled SIGKILL. To construct this mechanism, SYGMA uses `ptrace_notify()` on the Seccomp trace path triggered when a Seccomp-BPF filter returns `SECCOMP_RET_TRACE`, as shown in Figure 5. Before application execution, the Agent attaches a tracer configured with `PTRACE_O_TRACESECCOMP`, the clone/fork/vfork trace options, and `PTRACE_O_EXITKILL`. The tracer merely resumes each Seccomp stop with `PTRACE_CONT`; policy decisions remain with the in-kernel Policy Enforcer. At `ptrace_notify()`, the Policy Enforcer collects the current task's PID and mount namespaces and thread ID, extracts the syscall number and arguments from saved registers, retrieves the thread state from the State Map, and evaluates the active state-machine policy. The kernel then checks the task's pending fatal-signal state at the *Fatal Signal Gate*. Although these points were not designed as a coordinated path, SYGMA pairs them because a pending SIGKILL cannot be caught or ignored [35] and persists until the gate checks it.

For an ALLOW decision, the Policy Enforcer leaves the fatal-signal state unchanged, completes the Ptrace notification, and continues to syscall-table lookup and actual syscall execution. For a DENY decision, it marks SIGKILL as pending in the current task. The subsequent Fatal Signal Gate detects this state and transfers control to the exit loop without proceeding to syscall-table lookup or actual syscall execution. The process then terminates without executing the policy-violating syscall. Appendix C details this control path at the kernel source level. Thus, the decision and its enforcement occur at separate points but form a single preemptive kernel control path without delegating the policy decision to user space. Activating the Ptrace path, however, introduces additional operations, including a trace stop, tracer scheduling, resumption, and context switching. Routing every syscall through this path would accumulate these transition costs on each invocation. SYGMA therefore reserves Ptrace-Coupled SIGKILL for syscalls that require it, based on the per-syscall classification described next.

Analysis-Guided Enforcement Routing. The four syscalls that Conventional SIGKILL can enforce preemptively, `clone()`, `clone3()`, `fork()`, and `vfork()`, also fall within the LSM-BPF coverage. In the evaluated kernel build, however, the fatal-signal check points for these syscalls precede the corresponding LSM enforcement points, and Conventional SIGKILL requires only a single probe at the syscall-entry tracepoint. SYGMA therefore applies Syscall-Entry SIGKILL to these four syscalls because it blocks them earlier with lower enforcement cost. It routes the remaining syscalls certified as preemptively enforceable by LSM-BPF to LSM-BPF and applies Ptrace-Coupled SIGKILL only to syscalls for which neither LSM-BPF nor Conventional SIGKILL establishes preemptive enforceability. This routing strategy exploits the coverage of existing kernel mechanisms while using Ptrace-Coupled SIGKILL to close the remaining coverage gap.

Selective Ptrace-Path Activation. SYGMA activates the Ptrace entry path only for syscalls that require Ptrace-Coupled SIGKILL. To this end, it installs a selective Seccomp filter on each protected process. Based on the Analyzer classification, the filter returns `SECCOMP_RET_TRACE` only for syscalls assigned to Ptrace-Coupled

SIGKILL. Figure 5 illustrates this routing for `setns()`: when the container invokes the syscall, the Seccomp filter directs it to the Ptrace path, where the Policy Enforcer runs at `ptrace_notify()`. Syscalls assigned to LSM-BPF or Syscall-Entry SIGKILL are excluded from the trace action and therefore bypass the Ptrace path. This selective activation avoids imposing Ptrace-path overhead on syscalls that existing mechanisms can enforce.

The required Seccomp filter should apply `SCMP_ACT_TRACE` selectively according to the Analyzer classification. However, the Kubernetes `seccompProfile` interface cannot directly express per-syscall `SCMP_ACT_TRACE` rules. To install the filter without modifying the application binary, SYGMA uses a wrapper that installs a process-local Seccomp filter. The wrapper executes before the original application binary, applies `SCMP_ACT_TRACE` to the syscalls assigned to Ptrace-Coupled SIGKILL, and then invokes the original application through `execve()`. Because the installed Seccomp filter persists across `execve()`, the original application inherits the classification-specific routing, and only the selected syscalls subsequently activate the Ptrace path.

6.2 In-Kernel State-Machine Policy Evaluation

SYGMA enforces a state-machine policy generated externally as a per-state allowlist. The policy defines allowed syscalls per state and transition rules that update state on specific syscalls, capturing both permissions and ordering constraints. Each thread maintains its own runtime state, initialized to the policy's initial state. On each syscall, the Policy Enforcer retrieves the thread's current policy state (or initializes it if absent) and checks the syscall against the current state's allowlist. If allowed, it then applies any matching transition rule to update the state; otherwise, the state is retained. Denied syscalls do not trigger state updates, ensuring state progression only reflects successful transitions.

The *State-Machine Policy Decision-Making* in Figure 5 illustrates this evaluation using `setns()` with syscall number 308. Each per-state allow set is stored as a 45-byte bitmap containing one bit for each of the 360 supported syscalls under a dense index. The policy map uses the PID namespace, mount namespace, and current state as its key. For raw syscall number n, the Policy Enforcer uses index $i = n$ for $n \leq 334$ and $i = n - 89$ for $n \geq 424$, omitting the unused range 335–423, and inspects byte $\lfloor i/8 \rfloor$ and bit $i \bmod 8$. For syscall 308, $i = 308$, so it selects byte offset $\lfloor 308/8 \rfloor = 38$ and examines bit $308 \bmod 8 = 4$. In the example, this bit is unset, indicating that `setns()` is denied in the current state. If the bit were set and execution allowed, the Policy Enforcer would consult a separate transition map keyed by the current state and syscall and update the thread state when a matching transition exists.

6.3 Stateful Initialization Policy Construction

Before deploying a protected container, SYGMA executes the same container twice to construct its stateful initialization policy. Both profiling runs use the same container image and configuration as the protected deployment and observe the invoked syscalls. The first run identifies the main binary that defines the boundary between initialization and main execution. Using this boundary, the second run collects per-thread syscall transitions from execution of the container entrypoint until immediately before execution of the main binary. Each pair of adjacent syscalls within a thread forms a transition in the initialization state machine. Rather than storing the syscall sequence, the profiler records only adjacent transitions for each thread. After storing the most recently observed syscall s_i for a thread, it sets the transition bit corresponding to (s_i, s_j) when the next syscall s_j occurs. It separately records the first syscall observed in each thread as a valid transition from the initial state.

Recording transitions from multiple threads in a shared bitmap can lose updates because of concurrent modification. Specifically, setting a bit in a bitmap word requires a read–modify–write operation; eBPF programs executing on different CPUs may read the same value concurrently and overwrite one update with another [7]. To avoid this race, SYGMA records observed transitions in a per-CPU map instead of a shared bitmap. A per-CPU map provides an value for each CPU and therefore avoids synchronization over a shared value, a property also used by prior eBPF systems to reduce concurrent-access costs [59]. After profiling, SYGMA constructs a single initialization policy by merging the bitmaps from all CPUs with bitwise OR over the common transition space. Partitioning the observations does not alter the transition set because the bitmap records only transition existence and OR is idempotent. The result is therefore independent of the CPU on which a transition was observed. Moreover, the number of bitmap copies scales only with the number of CPUs, and each copy has a fixed size determined by the syscall space. Memory usage thus remains constant regardless of profiling duration or the number of observed syscall invocations.

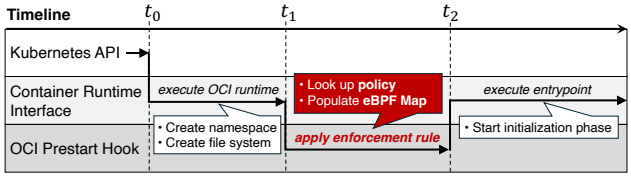


Figure 6: Lifecycle synchronization of initialization-policy activation with container startup. The OCI `prestart` hook installs the policy and initializes its state before the container entrypoint executes.

6.4 Lifecycle-Synchronized Policy-State Management

SYGMA uses Open Container Initiative (OCI) [47] hooks to synchronize policies and runtime state with container creation, the init-to-main transition, and termination. Before startup, it ensures that both the initialization and main-execution policies are available on the target node. At container creation, the `prestart` hook runs after namespace and filesystem setup but before the container entrypoint, as shown in Figure 6. It loads both policies into the eBPF maps, activates the initialization policy, and initializes the runtime state. The container runtime resumes only after these operations complete, ensuring that the entrypoint and early syscalls execute under the initialization policy.

During initialization, the initialization policy remains active until the kernel observes the `execve()` event for the identified main binary. Upon this event, SYGMA atomically activates the main-execution policy and reinitializes the runtime state through an

Table 3: Summary of evaluated applications and microservice workloads.

Application	Type	#Containers	Research Question
Ubuntu [10]	OS	1	RQ1
Nginx [45]	Web	1	RQ2, RQ4
Httpd [3]	Web	1	RQ2, RQ4
MySQL [48]	Database	1	RQ2, RQ4
Redis [54]	Key-Value Store	1	RQ2, RQ4
Hotel Reservation [14]	Microservice	19	RQ3
Online Boutique [46]	Microservice	11	RQ3

eBPF-map update, preventing syscalls from being evaluated against mismatched policy and state. At termination, the `poststop` hook uses the container identifier to remove the policies and thread-level runtime state. Because a restart or update produces a termination followed by a creation, SYGMA removes the previous instance's state and installs the policies and initial state anew, preventing stale state from persisting or being associated with a reused identifier.

7 Evaluation

We evaluate SYGMA through the following four research questions:

- RQ1. (*Enforceability*) Does SYGMA block a policy-violating syscall during execution before it returns successfully? (Section 7.1)
- RQ2. (*Precision*) How much does the initialization policy of SYGMA reduce the attack surface while reliably permitting normal container startup? (Section 7.2)
- RQ3. (*Consistency*) Does SYGMA eliminate the policy activation gap at container startup by synchronizing policies and runtime state with lifecycle events? (Section 7.3)
- RQ4. (*Performance*) How efficiently does SYGMA operate on server applications relative to existing enforcement mechanisms? (Section 7.4)

Implementation. We implemented SYGMA in approximately 15K lines of C and Go code for Linux kernel 5.15. The design considerations in Section 4.2 are realized through kernel-level enforcement and lifecycle-integrated policy management. The Agent registers OCI hooks through an NRI (Node Resource Interface) [38] plugin and uses them to configure policies and runtime state in eBPF maps when containers start or terminate. The binary wrapper installs a Seccomp filter through `prctl()` and then invokes the original entrypoint without modifying the container image or target binary.

Experimental Setup. We evaluated SYGMA on a single-node Kubernetes v1.29.1 cluster with an Intel Xeon Gold 6342 processor operating at 2.80 GHz and 32 GB of memory, running Linux kernel v5.15 and containerd v1.7.24 [13]. The single-node configuration avoids variability from inter-node communication and distributed coordination, allowing the measurements to primarily capture the costs of the evaluated security mechanisms. As summarized in Table 3, the evaluation covers standalone server applications, real-world microservice benchmarks, and a proof of concept (PoC) for a known vulnerability. We use *Tetragon* [64], a mature eBPF-based runtime security platform, as the comparison baseline.

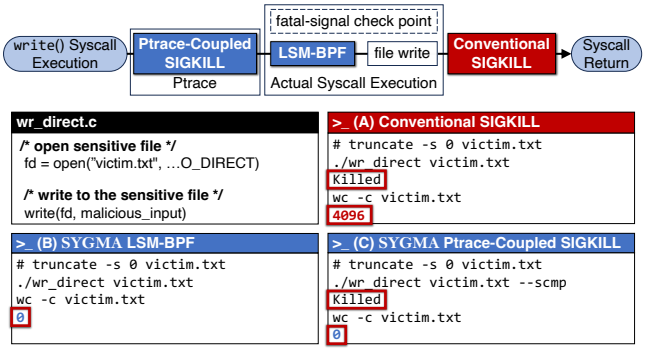


Figure 7: Preemptive enforcement of `write()`: Conventional SIGKILL (A) allows file modification before termination, whereas LSM-BPF (B) and Ptrace-Coupled SIGKILL(C) prevent it.

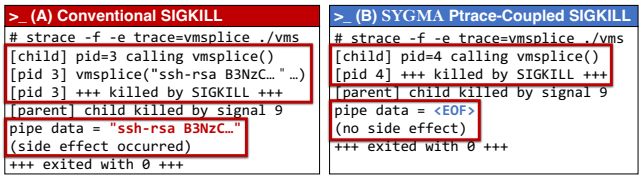


Figure 8: Preemptive enforcement of `vmsplice()`: Conventional SIGKILL (A) terminates the process after pipe insertion, whereas Ptrace-Coupled SIGKILL (B) blocks the syscall beforehand.

7.1 Enforcement Adequacy (RQ1)

We evaluate whether SYGMA blocks a policy-violating syscall during execution before it returns successfully. Because the blocking point depends on the enforcement path assigned to each syscall by the Analyzer, we test `write()` and `vmsplice()`, which Section 5 assigns to different paths. For these cases, after each violating process terminates, we additionally inspect persistent state changes in the affected kernel object to determine whether the observed blocking point precedes the policy-relevant side effect.

Use Cases. To validate the preemptive enforcement of SYGMA, we evaluate `write()` and `vmsplice()`, which the Analyzer assigns to different paths. The Analyzer classifies `write()` as *Preemptively Enforceable* by LSM-BPF and *Non-Enforceable* by Conventional SIGKILL, and therefore assigns it to the LSM-BPF path. Although SYGMA routes `write()` to LSM-BPF, we also evaluate Ptrace-Coupled SIGKILL to compare the enforcement timing of all three mechanisms. It classifies `vmsplice()` as *Non-Enforceable* by both Conventional SIGKILL and LSM-BPF, and therefore assigns it to the Ptrace-Coupled SIGKILL path. We conduct both experiments in a protected container based on an Ubuntu image and apply a policy that denies only the target syscall. To compare enforcement timing, we apply each mechanism independently and inspect the kernel object after the violating process terminates.

In the `write()` experiment, we empty a target file, open it with `O_DIRECT`, attempt a write in a policy-violating state, and measure

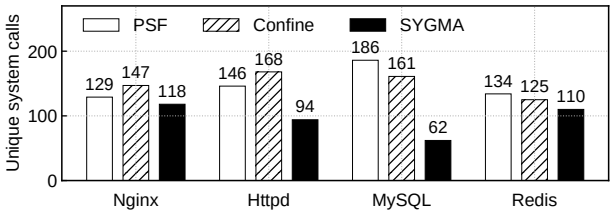


Figure 9: Number of unique syscalls permitted by the initialization policies.

Table 4: Availability and transition stability of initialization policies across 100 repeated deployments.

Workload	False Rejection Rate	Stable Transitions
Nginx	6.0%	99.6%
Httpd	0.0%	100.0%
MySQL	2.0%	98.8%
Redis	5.0%	99.1%

its size after termination. As shown in Figure 7, Ptrace-Coupled SIGKILL reaches a fatal-signal check before syscall execution, while LSM-BPF denies the write at an LSM hook. In contrast, the `write()` path has no fatal-signal check before the write, preventing Conventional SIGKILL from blocking it preemptively. Consequently, Conventional SIGKILL leaves 4,096 bytes in the file, whereas LSM-BPF and Ptrace-Coupled SIGKILL leave it empty. This demonstrates the limitation of post-execution termination in Figure 2.

The `vmsplice()` syscall connects user-space pages to a pipe buffer, allowing another process to read them even after the caller terminates. Such pipe-buffer manipulation is also central to attacks such as Dirty Pipe [33]. In the experiment, a child inserts sensitive data into a pipe using `vmsplice()` and terminates, after which the parent attempts to read the data. We use `SPLICE_F_NONBLOCK` to observe only the non-waiting insertion path. Because `vmsplice()` has no applicable LSM hook, we compare Conventional SIGKILL and Ptrace-Coupled SIGKILL. As shown in Figure 8, Conventional SIGKILL terminates the child after page insertion, allowing the parent to read the data. With Ptrace-Coupled SIGKILL, `strace` shows no entry into `vmsplice()`, and the parent's read returns EOF. Thus, although both mechanisms terminate the child, only preemptive enforcement prevents the policy-relevant side effect.

In both cases, LSM-BPF and Ptrace-Coupled SIGKILL block violations on different kernel execution paths before the observed policy-relevant side effects occur. In particular, the `vmsplice()` result demonstrates that some syscalls are not preemptively enforceable by either mechanism and that Ptrace-Coupled SIGKILL closes this coverage gap. The observed results for both syscalls match the Analyzer's prior classification, supporting that analysis-guided routing selects the intended enforcement boundary at runtime.

7.2 Initialization Policy Precision and Reliability (RQ2)

Attack Surface Reduction. We compare the syscall attack surface permitted by the initialization policy against PSF [11] and Confine [21]. As shown in Figure 9, SYGMA permits 62–118 syscalls depending on the workload, whereas PSF and Confine permit 125–186. This corresponds to reductions of 9–67% relative to PSF and 12–61% relative to Confine. In addition to reducing the set of permitted syscalls, SYGMA constrains 80–281 transitions among them and thus blocks a permitted syscall when it occurs in an order undefined by the policy. Statistical estimation in PSF can leave overly permissive enforcement intervals, while the static policy of Confine retains privileges required for initialization during subsequent execution. Both approaches encode their analysis results as stateless allowlists and therefore cannot constrain syscall ordering. In contrast, SYGMA constructs a stateful policy from transitions observed during initialization, providing a more restrictive least-privilege boundary until main execution begins.

Initialization Policy Reliability. SYGMA constructs its initialization policy from transitions observed during a single profiling run. We therefore evaluate whether this policy reliably permits normal deployments and whether the collected transitions recur consistently across deployments. We fix the policy generated for each workload and deploy the application 100 times, for 400 deployments in total, recording both policy misses and all transitions that actually occur. We define the deployment-level false rejection rate as the fraction of deployments with at least one initialization-policy miss. We define transition stability as the fraction of policy transitions that recur in all 100 deployments.

As shown in Table 4, the false rejection rate ranges from 0–6% across workloads, with no misses for Httpd. Among the transitions included in each policy, 98.8–100% recur in every deployment. Both the missing recurrences and the policy misses across all 400 deployments trace back to a single transition into `arch_prctl()`, issued when a new thread or process is spawned during initialization. Whether this transition falls before or after the phase boundary is inherently nondeterministic, so a single profiling run does not always observe it. The misses therefore do not indicate scattered gaps across unrelated initialization paths, and SYGMA still permits at least 94% of normal deployments for every workload despite enforcing a substantially narrower boundary than existing approaches.

7.3 Lifecycle-Synchronized Policy Activation (RQ3)

We evaluate whether SYGMA activates a policy before container entrypoint execution begins by comparing it with Tetragon's asynchronous Operator-based activation. Without a runtime hook, Tetragon obtains the container's cgroup information through a Kubernetes Operator and then registers the container as a policy target. Both mechanisms preload the policy, so the measurements exclude policy loading time and capture only the policy-state configuration delay.

We evaluate 30 containers, comprising 19 from Hotel Reservation and 11 from Online Boutique, including application and infrastructure services such as Consul, Jaeger, MongoDB, and Memcached.

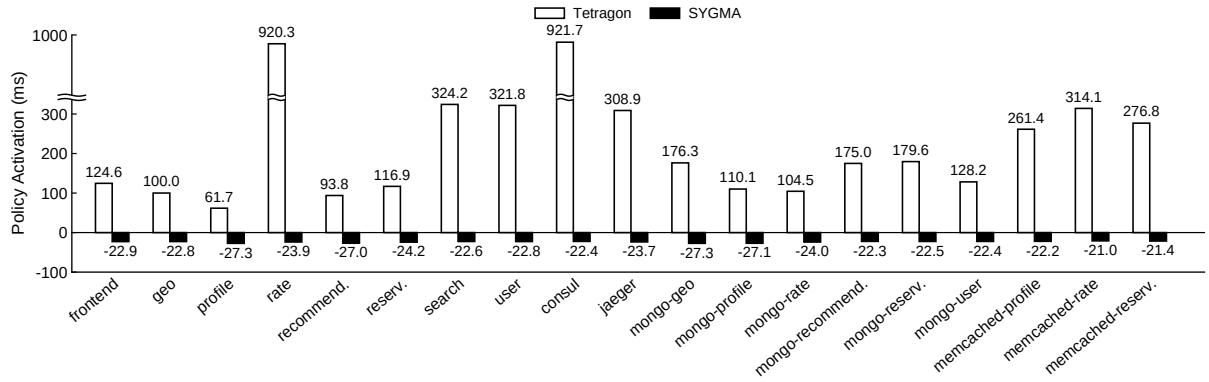


Figure 10: Policy activation times for the 19 containers in Hotel Reservation. Each value is the median of 10 trials relative to container entrypoint execution at time zero. Negative values for SYGMA **indicate activation before the entrypoint, whereas positive values for Tetragon indicate activation after the entrypoint.**

Table 5: Syscalls executed before policy activation and thus missed by Tetragon. Each value is the workload-wide mean of the per-container medians. For SYGMA**, both metrics were zero for every container.**

Microservice Application	Syscall Invocations		Unique Syscalls	
	Tetragon	SYGMA	**Tetragon**	SYGMA
Hotel Reservation	1,757.1	0	55.1	0
Online Boutique	8,535.7	0	45.3	0

For each container-mechanism pair, we perform two warm-up trials followed by 10 measured trials, recording the interval between policy activation and the first syscall and the number of syscalls executed during that interval. We use the `execve()` of container PID 1 as the reference point because no in-container process exists for tracing before it, and an independent eBPF observer tracks syscalls from both PID 1 and its child processes.

Figure 10 shows the activation times for Hotel Reservation, while Online Boutique exhibits a similar trend (Appendix D). Across all 30 containers, SYGMA activates the policy a median of 21.00–27.30 ms before the entrypoint. This ordering holds in every trial, with no syscall observed before policy activation. In contrast, Tetragon completes cgroup registration after application execution begins, resulting in a median post-first-syscall activation gap ranging from 61.68 ms for `hotel-profile` to 921.70 ms for `hotel-consul`. As shown in Table 5, Tetragon leaves an average of 1,757.1 and 8,535.7 syscalls per container unenforced for Hotel Reservation and Online Boutique, respectively, spanning 55.1 and 45.3 unique syscalls, whereas SYGMA misses none in either workload.

This activation gap arises because container identification is delivered through a control-plane path decoupled from application execution. An asynchronous mechanism must wait for the container identifier before registering a new process as a policy target, allowing the application to execute first. In contrast, SYGMA configures the policy and runtime state in the `prestart` hook and suspends container execution until this operation completes. Policy activation therefore precedes container execution regardless of the application implementation. More generally, the same synchronization issue can arise whenever a policy or state update is decoupled from the lifecycle event it governs. SYGMA therefore binds creation, the init-to-main switch, and termination to the `prestart` hook, the main-binary `execve()` event, and the `poststop` hook, respectively, as described in Section 6.4. This experiment directly quantifies the effect of this synchronization at container creation; the other lifecycle transitions use the same event-coupled design but are not separately timed.

7.4 Runtime Performance Overhead (RQ4)

Benchmark Configuration. We measure the runtime overhead of SYGMA through repeated benchmarks of real application workloads. We run each experiment 100 times and report the mean, normalizing the results relative to a baseline without a security mechanism to compare heterogeneous workloads. We define performance degradation as the latency increase for latency-based applications and the throughput reduction for throughput-based applications. All experiments use a policy that permits the syscall transitions observed during normal execution. Because a denied syscall terminates the corresponding request and prevents benchmark completion, we permit all observed transitions to measure enforcement costs under a normal workload.

For Nginx and Httpd, Apache Bench v2.3 [2] sends 500 requests of 64–512 KB with a concurrency of 100. For MySQL, we run four sysbench v1.0.20 [62] workloads (`read_write`, `insert`, `delete`, and `point_select`) with 10 threads. For Redis, memtier_benchmark [53] performs 2M operations across multiple data sizes. We compare Tetragon [64], SYGMA, and SYGMA-all-trace, which disables Analysis-Guided Enforcement Routing and applies `SECCOMP_RET_TRACE` to every syscall to measure the cost of Ptrace-path transitions.

Runtime Performance. As shown in Figure 11, SYGMA incurs 1.8–5.1% performance degradation, lower than Tetragon's 8.8–21.1% across all workloads. These measurements include state-transition evaluation in addition to syscall permission checks. In contrast, SYGMA-all-trace, which uses the same policy and Policy Enforcer,

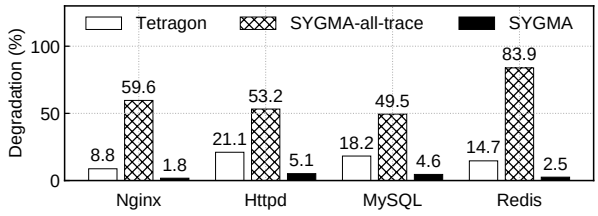


Figure 11: Performance degradation relative to the baseline. We use the latency increase for latency-based workloads and the throughput reduction for Redis. SYGMA-**all-trace is a comparison configuration that routes every syscall through the Ptrace path.**

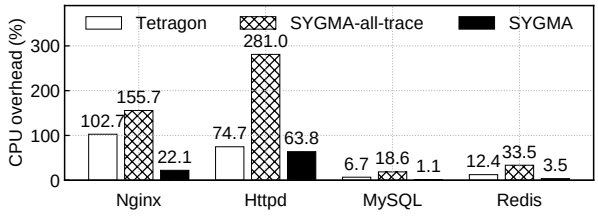


Figure 12: Increase in CPU usage relative to the baseline, measured from the cgroup of each workload container.

incurs 49.5–83.9% degradation. Because the only difference between the two configurations is per-syscall Ptrace-path activation, this gap results from trace stops and the associated scheduling and context-switch costs. Routing every syscall through the Ptrace path accumulates these costs in syscall-intensive workloads. Analysis-Guided Enforcement Routing restricts this path to syscalls that require Ptrace-Coupled SIGKILL, limiting performance degradation while retaining preemptive coverage.

CPU Overhead and Routing Ratio. We measure CPU usage from each workload container's cgroup. Figure 12 shows that SYGMA incurs lower CPU overhead than Tetragon across all workloads, depending on the LSM-BPF/Ptrace routing ratio. For MySQL and Redis, LSM-BPF handles over 95% of syscalls, limiting overhead to 1.1% and 3.5%, respectively. Nginx and Httpd route more syscalls through Ptrace, with Httpd reaching a 75.03% Ptrace ratio and 63.8% overhead, still below Tetragon due to LSM-BPF routing of the remaining syscalls. SYGMA-all-trace incurs the highest overhead, reaching 281.0% for Httpd, over four times that of SYGMA. Since both configurations share the same Policy Enforcer, the difference demonstrates that Analysis-Guided Enforcement Routing improves response performance and CPU efficiency by limiting Ptrace to syscalls that require it.

8 Related Work

Syscall Policy Derivation and Sequence Modeling. Prior work derives stateless syscall policies for Linux applications and containers through static or dynamic analysis [9, 15, 21, 49, 70]. Phase-aware approaches apply separate Seccomp profiles to initialization and main execution [22, 23, 39, 52], while Timeloops [50] automates profiles for interpreted applications. However, these policies rely on `Seccomp-BPF` and cannot constrain state-dependent syscall ordering. Other work models syscall sequences for anomaly detection [16, 18, 29, 31, 36, 51, 69]. NIMOS [60] extracts attack sequences from exploit code, while B-Side [68] represents per-phase syscall flows as deterministic finite automata. Unlike these approaches, SYGMA enforces state-machine policies in the kernel and extends stateful protection to container initialization.

Syscall Enforcement Mechanisms. SysXCHG [19] dynamically exchanges syscall filters during `execve`, but still applies stateless allow/deny policies. Phoenix [34] combines Seccomp filtering with Ptrace-based inspection to enforce malicious syscall-sequence blocklists, while Sifter [30] uses eBPF allowlists to protect security-critical Android kernel modules. Existing eBPF enforcement mechanisms also face timing and coverage limitations. Tetragon [64] and BPFGuard [41] deliver `SIGKILL` after detecting violations during syscall execution, which does not guarantee blocking before completion. LSM-BPF provides preemptive enforcement only when applicable hooks occur on every relevant execution path and their denials propagate to syscall failure [28], while `bpf_override_return`-based enforcement requires additional kernel configuration [56]. In contrast, SYGMA systematically analyzes per-syscall execution paths and enforcement boundaries, then routes each syscall to Syscall-Entry SIGKILL, LSM-BPF, or Ptrace-Coupled SIGKILL. For syscalls that existing mechanisms cannot enforce preemptively, it combines `ptrace_notify()` with standard eBPF functionality to close the coverage gap without kernel modifications or nonstandard configurations.

9 Discussion

Inherited Determinism Assumptions and Policy Refinement. The state-machine policy in SYGMA models execution as syscall-level state transitions, inheriting its inherent limitations. User-space control-flow decisions (e.g., C++ exceptions, JVM dynamic loading, green-thread scheduling) are not directly observable at the syscall layer and only manifest through subsequent syscalls. SYGMA handles kernel-visible nondeterminism separately; for signals, it uses the `signal_deliver` tracepoint to isolate handler execution as a subgraph, avoiding false policy violations. When unobservable control flow leads to apparent violations, SYGMA records the state and transition as evidence and reports it to the administrator, who can update the policy at runtime without restarting the container.

Policy and Kernel Extensibility. SYGMA can be extended to support policies more fine-grained than its current representation and to accommodate different kernel builds. The Policy Enforcer currently evaluates policies using syscall numbers and state transitions, whereas the enforcement paths act only on the resulting policy decisions. Thus, policy evaluation can incorporate syscall arguments or additional execution context within the constraints of the eBPF verifier without changing the per-syscall enforcement-path assignments or preemptive blocking mechanisms. Because the Analyzer uses the target kernel's source code and build configuration as inputs, the same analysis procedure can be applied to other kernel versions or build configurations to generate a syscall

classification for each build. SYGMA configures the enforcement path for each syscall according to this classification.

10 Conclusion

In this paper, we presented SYGMA, a stateful syscall enforcement framework for containerized environments. We systematically analyzed syscall execution paths and enforcement points in the Linux kernel and, guided by this analysis, selected the enforcement mechanism capable of preemptively blocking each syscall during execution before it returns successfully. SYGMA also extends stateful protection across the container lifecycle, from initialization through main execution, and synchronizes policies and runtime state with container creation and termination events. Evaluation across diverse workloads shows low overhead and no startup activation gap, demonstrating practical preemptive stateful enforcement without kernel modification.

References

[1] Mashal Abbas, Shahpar Khan, Abdul Monum, Fareed Zaffar, Rashid Tahir, David Eyers, Hassaan Irshad, Ashish Gehani, Vinod Yegneswaran, and Thomas Pasquier. 2022. Paced: Provenance-based automated container escape detection. In 2022 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 261–272.

[2] Apache Software Foundation. 2026. Apache HTTP Server Benchmarking Tool (ab). https://httpd.apache.org/docs/current/programs/ab.html.

[3] Apache Software Foundation. 2026. Apache HTTP Server Project. https://httpd.apache.org/.

[4] William Blair, Frederico Araujo, Teryl Taylor, and Jiyong Jang. 2024. Automated synthesis of effect graph policies for microservice-aware stateful system call specialization. In 2024 IEEE Symposium on Security and Privacy. IEEE, 4554–4572.

[5] Christian Brauner. 2020. The Seccomp Notifier – New Frontiers in Unprivileged Container Development. https://brauner.github.io/2020/07/23/seccomp-notify.html.

[6] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. 2021. Saphire: Sandboxing {PHP} applications with tailored system call allowlists. In 30th USENIX Security Symposium (USENIX Security 21). 2881–2898.

[7] Matthew Butrovich, Samuel Arch, Wan Shen Lim, William Zhang, Jignesh M Patel, and Andrew Pavlo. 2025. Bpf-db: A kernel-embedded transactional database management system for ebpf applications. Proceedings of the ACM on Management of Data 3, 3 (2025), 1–27.

[8] Claudio Canella, Sebastian Dorn, Daniel Gruss, and Michael Schwarz. 2022. Sfip: Coarse-grained syscall-flow-integrity protection in modern systems. arXiv preprint arXiv:2202.13716 (2022).

[9] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating seccomp filter generation for linux applications. In Proceedings of the 2021 on Cloud Computing Security Workshop. 139–151.

[10] Canonical Ltd. 2026. Ubuntu: Enterprise Open Source and Linux. https://ubuntu.com/.

[11] Ke Chen, Hui Lu, Yinnan Yao, Binxing Fang, Yuan Liu, and Zhihong Tian. 2025. Enhancing Container Security through Phase-Based System Call Filtering. IEEE Transactions on Cloud Computing (2025).

[12] Theo Combe, Antony Martin, and Roberto Di Pietro. 2016. To docker or not to docker: A security perspective. IEEE Cloud Computing 3, 5 (2016), 54–62.

[13] containerd Authors. 2026. containerd: An Industry-Standard Container Runtime. https://containerd.io/.

[14] deathstarbench 2024. DeathStarBench. https://github.com/delimitrou/DeathStarBench

[15] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. 2020. Sysfilter: Automated system call filtering for commodity software. In 23rd International Symposium on Research in Attacks, Intrusions and Defenses. 459–474.

[16] Henry Hanping Feng, Jonathon T Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P Miller. 2004. Formalizing sensitivity in static analysis for intrusion detection. In IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004. IEEE, 194–208.

[17] William Findlay, Anil Somayaji, and David Barrera. 2020. Bpfbox: Simple precise process confinement with ebpf. In Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop. 91–103.

[18] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. 1996. A sense of self for unix processes. In Proceedings 1996 IEEE symposium on security and privacy. IEEE, 120–128.

[19] Alexander J Gaidis, Vaggelis Atlidakis, and Vasileios P Kemerlis. 2023. Sysxchg: Refining privilege with adaptive system call filters. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. 1964–1978.

[20] Tal Garfinkel et al. 2003. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In NDSS, Vol. 3. 163–176.

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

[22] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal system call specialization for attack surface reduction. In 29th USENIX Security Symposium. 1749–1766.

[23] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2c: Fine-grained configuration-driven system call filtering. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. 1243–1257.

[24] Songi Gwak, Thien-Phuc Doan, and Souhwan Jung. 2023. Container Instrumentation and Enforcement System for Runtime Security of Kubernetes Platform with eBPF. Intelligent Automation & Soft Computing 37, 2 (2023).

[25] Md Sadun Haq, Thien Duc Nguyen, Ali Şaman Tosun, Franziska Vollmer, Turgay Korkmaz, and Ahmad-Reza Sadeghi. 2024. SoK: A comprehensive analysis and evaluation of docker container attack and defense mechanisms. In 2024 IEEE Symposium on Security and Privacy. IEEE, 4573–4590.

[26] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. 2023. Cross container attacks: The bewildered {eBPF} on clouds. In 32nd USENIX Security Symposium (USENIX Security 23). 5971–5988.

[27] Jin Her, Chihyeon Cho, Jaehyun Nam, and Seungsoo Lee. 2026. Beyond Seccomp: Breaking and Rebuilding Syscall Filtering for Microservices. Black Hat USA 2026 Briefings. https://blackhat.com/us-26/briefings/schedule/index.html#beyond-seccomp-breaking-and-rebuilding-syscall-filtering-for-microservices-52672

[28] Jin Her, Chi Hyeon Jo, Taejune Park, and Seungsoo Lee. 2024. KUBEROSY: A Dynamic System Call Filtering Framework for Containers. IEEE Access (2024).

[29] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. 1998. Intrusion detection using sequences of system calls. Journal of computer security 6, 3 (1998), 151–180.

[30] Hsin-Wei Hung, Yingtong Liu, and Ardalan Amiri Sani. 2022. Sifter: protecting security-critical kernel modules in Android through attack surface reduction. In Proceedings of the 28th Annual International Conference on Mobile Computing and Networking. 623–635.

[31] Kapil Jain and R Sekar. 2000. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In Network and Distributed Systems Security Symposium.

[32] Jinghao Jia, Yifei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. 2023. Programmable System Call Security with eBPF. CoRR abs/2302.10366 (2023).

[33] Max Kellermann. 2022. The Dirty Pipe Vulnerability. https://dirtypipe.cm4all.com/.

[34] Hugo Kermabon-Bobinnec, Yosr Jarraya, Lingyu Wang, Suryadipta Majumdar, and Makan Pourzandi. 2024. Phoenix: Surviving Unpatched Vulnerabilities via Accurate and Efficient Filtering of Syscall Sequences. In Network and Distributed Systems Security Symposium.

[35] Michael Kerrisk. 2026. signal(7) — Linux Manual Page. https://man7.org/linux/man-pages/man7/signal.7.html.

[36] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2005. Automating mimicry attacks using static binary analysis. In USENIX Security Symposium, Vol. 14. 161–176.

[37] kubearmor 2026. KubeArmor - Cloud Native Runtime Security Enforcement Engine. https://kubearmor.io/

[38] Kubernetes SIG Node. 2025. Node Resource Interface (NRI). https://github.com/containerd/nri.

[39] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. Speaker: Split-phase execution of application containers. In International Conference on Detection of intrusions and malware, and vulnerability assessment. Springer, 230–251.

[40] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html.

[41] Hui Lu, Xiaojiang Du, Dawei Hu, Shen Su, and Zhihong Tian. 2025. BPF-Guard: Multi-Granularity Container Runtime Mandatory Access Control. IEEE Transactions on Cloud Computing (2025).

[42] Linux man-pages project. 2023. ptrace(2) — Linux Manual Page. https://man7.org/linux/man-pages/man2/ptrace.2.html

[43] Rory McCune. 2022. Container Escape – CVE-2022-0185. https://blog.aquasec.com/cve-2022-0185-linux-kernel-container-escape-in-kubernetes/

[44] Eric Mountain, Christophe Tafani-Dereeper, Tommy McCormick, and Frederic Baguelin. 2022. Container Escape with CVE-2022-0847. https://securitylabs.datadoghq.com/articles/dirty-pipe-container-escape-poc/.

[45] NGINX, Inc. 2026. NGINX Official Website. https://nginx.org/.
[46] online-boutique 2026. microservices-demo. https://github.com/GoogleCloudPlatform/microservices-demo
[47] Open Container Initiative. 2025. Open Container Initiative Runtime Specification: Configuration. https://github.com/opencontainers/runtime-spec/blob/main/config.md.
[48] Oracle Corporation. 2026. MySQL Official Website. https://www.mysql.com/.
[49] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated policy synthesis for system call sandboxing. Proceedings of the ACM on Programming Languages 4, OOPSLA (2020), 1–26.
[50] Meghna Pancholi, Andreas D Kellas, Vasileios P Kemerlis, and Simha Sethumadhavan. 2022. Timeloops: Automatic system call policy learning for containerized microservices. arXiv preprint arXiv:2204.06131 (2022).
[51] Chetan Parampalli, R Sekar, and Rob Johnson. 2008. A practical mimicry attack against powerful system-call monitors. In Proceedings of the 2008 ACM symposium on Information, computer and communications security. 156–167.
[52] Vidya Lakshmi Rajagopalan, Konstantinos Kleftogiorgos, Enes Göktas, Jun Xu, and Georgios Portokalidis. 2023. Syspart: Automated temporal system call filtering for binaries. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. 1979–1993.
[53] Redis Ltd. 2026. memtier_benchmark: A High-Throughput Benchmarking Tool for Redis and Memcached. https://github.com/RedisLabs/memtier_benchmark.
[54] Redis Ltd. 2026. Redis Official Website. https://redis.io/.
[55] Michael Reeves, Dave Jing Tian, Antonio Bianchi, and Z Berkay Celik. 2021. Towards improving container security by preventing runtime escapes. In 2021 IEEE Secure Development Conference (SecDev). IEEE, 38–46.
[56] Matthew Rossi, Michele Beretta, Dario Facchinetti, and Stefano Paraboschi. 2025. POSTER: Transparent Temporally-Specialized System Call Filters. In Proceedings of the 20th ACM Asia Conference on Computer and Communications Security. 1797–1799.
[57] seccomp 2023. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html
[58] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. 2001. A fast automaton-based method for detecting anomalous program behaviors. In Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001. IEEE, 144–155.
[59] R Sekar, Hanke Kimm, and Rohit Aich. 2024. eaudit: A fast, scalable and deployable audit data collection system. In 2024 IEEE Symposium on Security and Privacy. IEEE, 3571–3589.
[60] Somin Song, Sahil Suneja, Michael V Le, and Byungchul Tak. 2023. On the value of sequence-based system call filtering for container security. In 2023 IEEE 16th International Conference on Cloud Computing. IEEE, 296–307.
[61] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. 2019. Container security: Issues, challenges, and the road ahead. IEEE access 7 (2019), 52976–52996.
[62] sysbench developers. 2020. sysbench: a system performance benchmark. https://github.com/akopytov/sysbench.
[63] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. 2017. Understanding security implications of using containers in the cloud. In 2017 USENIX Annual Technical Conference (USENIX ATC 17). 313–319.
[64] tetragon 2024. Tetragon - eBPF-based Security Observability Runtime Enforcement. https://tetragon.io/
[65] The kernel development community. 2015. Control Groups. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.
[66] The kernel development community. 2019. Namespaces. https://www.kernel.org/doc/html/latest/admin-guide/namespaces/index.html.
[67] The Kubernetes Authors. 2024. Overview of Kubernetes. https://kubernetes.io/docs/concepts/overview/.
[68] Gaspard Thévenon, Kevin Nguetchouang, Kahina Lazri, Alain Tchana, and Pierre Olivier. 2024. B-Side: Binary-Level Static System Call Identification. In Proceedings of the 25th International Middleware Conference. 225–237.
[69] David Wagner and R Dean. 2001. Intrusion detection via static analysis. In Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001. IEEE, 156–168.
[70] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. 2017. Mining sandboxes for linux containers. In 2017 IEEE International Conference on Software Testing, Verification and Validation. IEEE, 92–102.
[71] Seungyong Yang, Brent Byunghoon Kang, and Jaehyun Nam. 2024. Optimus: association-based dynamic system call filtering for container attack surface reduction. Journal of Cloud Computing 13, 1 (2024), 71.

Open Science

To support reproducibility and independent audit, we provide an anonymized artifact at https://github.com/cclab-inu/SYGMA. The artifact includes the source code and build and deployment instructions for the SYGMA Analyzer and runtime framework, together with the Linux 5.15 x86-64 per-syscall classifications, runtime routes, and supporting evidence for the *Enforceable*, *Bypass-Certified*, and *Unresolved* decisions; the latter two are jointly reported as *Non-Enforceable* in the paper.

Listing 2: Example of a `SYGMAConfig` custom resource

```
apiVersion: security.security.io/v1
kind: SYGMAConfig
metadata:
  name: nginx-config
  namespace: default
spec:
  collection:
    yamlContent: |
      apiVersion: v1
      kind: Pod
      metadata:
        name: nginx
        namespace: default
        labels:
          app: nginx
      spec:
        containers:
        - name: nginx
          image: nginx:1.24.0
          imagePullPolicy: Always
    mainExecutionPhase:
      outputContent: |
        {
          "startState": "INIT",
          "states": {
            "INIT": {
              "allowedSyscalls": ["brk","mmap","prctl"],
              "transitions": {
                "brk": "INIT", "mmap": "INIT",
                "prctl": "EVENT_LOOP"
              }
            },
            "EVENT_LOOP": {
              "allowedSyscalls": ["epoll_wait","accept4"],
              "transitions": {
                "epoll_wait": "EVENT_LOOP",
                "accept4": "HANDLING_CONN"
              }
            }
          }
        }
```

A SYGMA Config Specification

SYGMA manages protected-container configurations through a Kubernetes Custom Resource Definition (CRD). An administrator deploys a `SYGMAConfig` custom resource to specify the workload to protect and the policy inputs to apply to that workload. The Operator watches this resource and coordinates policy preparation, transformation, and deployment for the specified workload.

The `spec` of `SYGMAConfig` consists of a `collection` field that specifies the inputs required for policy construction. `collection.yamlContent` directly embeds the Pod definition of the protected workload. The Operator extracts the container image and execution configuration from this definition to identify the target for initialization profiling. `collection.mainExecutionPhase.outputContent` provides the main-execution policy generated by an external generator. This policy consists of `startState`, which specifies the initial state, and `states`, which contains the definition of each state. Each state defines the set of syscalls allowed in that

state (`allowedSyscalls`) and transition rules that specify the next state upon invocation of a particular syscall (`transitions`). In Listing 2, `INIT` is the initial state of the externally supplied main-execution policy and is distinct from the initialization policy that SYGMA constructs separately. This state permits the syscalls required while the main binary prepares its execution environment and transitions to `EVENT_LOOP` upon `prctl`. `EVENT_LOOP` then permits only the syscalls required to wait for requests and transitions to a connection-handling state upon `accept4`. The Operator's Policy Preprocessor converts this representation into a bitmap-based allowlist and a state-transition map, and the Agent installs the resulting runtime representation in the Filter before container execution.

B SYGMA Analyzer Design and Implementation

This appendix details the implementation of the SYGMA Analyzer, the offline static-analysis component that produces the per-syscall classifications used in Section 5. For each target kernel build, the Analyzer consumes the kernel source, build configuration, compilation database, and binary artifacts. It outputs the enforcement classification used by the runtime Policy Enforcer together with mechanism-specific evidence: an E certificate, an N bypass witness, or a U unresolved gap. The analysis is performed once per kernel build and is independent of runtime enforcement.

B.1 Analysis Frontend

The Analyzer first restricts its input to code included in the target build. Kernel sources contain configuration-dependent code that may not be compiled or executable under a particular configuration. Analyzing such code could incorrectly treat an unreachable LSM hook or signal check as an enforcement point. The Analyzer therefore reads the `compile_commands.json` generated during the target-kernel build and applies the recorded compiler options and preprocessor definitions to each translation unit.

The frontend uses Clang to extract the Abstract Syntax Tree (AST) and Control-Flow Graph (CFG) of each function. It constructs interprocedural call relations, propagates call arguments, and maps `security_*` hooks and signal checks such as `fatal_signal_pending()` to their CFG blocks. Constant call arguments are used to eliminate unreachable branches. If a call site cannot be mapped uniquely to a CFG block, the Analyzer records it as unplaced rather than assigning it arbitrarily. A path containing an unplaced call remains unresolved unless the Analyzer separately establishes that the call cannot affect the classification.

Source-level control flow alone cannot always determine how a hook's return value reaches the final syscall return. Inlining and compiler optimization may transform value propagation across function boundaries. The Analyzer therefore inspects LLVM IR for only the translation units involved in the current syscall classification. It tracks whether a denial value survives optimization and reaches the syscall return as an error. A value discarded or converted to success invalidates the corresponding blocking evidence, whereas an unresolved propagation relation produces U. The Analyzer also recognizes recurring kernel error-propagation patterns, such as `if (err) return err;`, and combines them with CFG-derived facts.

B.2 Deriving Successful Return Candidates

Starting from each syscall entry function, the Analyzer follows interprocedural call relations to construct the successful return candidates defined in Section 5.1. A candidate represents a path that can reach a normal successful return. Paths proven to terminate through panic, debugging, or error-cleanup logic are excluded. The construction applies three conservative restrictions:

- **Return-value use.** A callee's denial value is accepted as blocking evidence only if it can affect the syscall result. If the value is unused, the call cannot establish blocking. If it is tested in a condition or passed to another function, the Analyzer follows its subsequent propagation before assigning a result.
- **Indirect calls.** Calls through function pointers or operation tables are followed only when their possible targets can be resolved statically. An unresolved indirect call that may contribute to a successful return keeps the candidate unresolved; the Analyzer does not assume that its target contains an enforcement point.
- **Recursive calls.** Mutually recursive functions are grouped into a cyclic call component. Candidate and classification states are propagated until a fixed point is reached. Any unresolved path remaining in the component is propagated to its callers.

These restrictions prevent incomplete call or value information from being interpreted as evidence of enforceability.

B.3 Enforceability Criteria

The Analyzer evaluates LSM-BPF and Conventional SIGKILL independently for every successful return candidate. Each candidate is assigned *Enforceable* (E), *Bypass-Certified* (N), or *Unresolved* (U). E proves that the mechanism blocks the candidate during syscall execution before the syscall returns successfully. N provides a concrete candidate that reaches successful return because no applicable enforcement point exists or because a denial is neutralized. U indicates that neither result can be established.

The Analyzer conservatively aggregates candidate-level results into a syscall-level classification. A syscall is classified as N if any candidate has a bypass witness; otherwise, it is classified as U if any candidate remains unresolved. Only when every successful return candidate is certified as blockable does the Analyzer classify the syscall as E. Section 5 reports E as *Preemptively Enforceable* and combines N and U as *Non-Enforceable*.

LSM-BPF. An LSM hook constitutes blocking evidence only if it has enforcement semantics, supports BPF attachment in the target build, and propagates denial to the final syscall result. The Analyzer excludes notification-only hooks and tracks each denial value through subsequent calls and branches. It distinguishes values that remain unchanged, are normalized or replaced by another error, are overwritten by success, or depend on an unresolved condition. A denial that may change between error values but cannot become success certifies the candidate as E. A path proven to restore success produces an N witness, whereas an unresolved overwrite or conditional propagation produces U. The resulting E certificate records the applicable hook and the error-propagation evidence for the candidate.

Conventional SIGKILL. A signal check is relevant only when it examines the task that invoked the syscall. The Analyzer therefore verifies at each call site that the checked task is `current`. It certifies a candidate only when the signal check occurs during syscall execution and a pending fatal signal eliminates the candidate's successful return path by transferring control to an error, abort, or kernel-exit path. Merely propagating the result of a signal check or evaluating it inside a loop is insufficient until the Analyzer proves that the final successful return is eliminated. Actual task termination then occurs on the syscall-exit path according to the kernel's pending-SIGKILL semantics. A signal check reached only in the exit loop after syscall execution is not accepted as preemptive blocking evidence.

For the process-creation syscalls classified as E, namely `clone()`, `clone3()`, `fork()`, and `vfork()`, the Analyzer additionally verifies that the certified signal check precedes task creation. A check reached only after a new task becomes visible is not accepted as preemptive blocking evidence.

B.4 Conservative Classification Principle

The implementation applies three fail-to-U conditions before emitting a conclusive classification:

- **Candidate completeness.** E requires blocking evidence for every successful return candidate. If only a subset is certified, the syscall remains U. Conversely, the absence of a discovered hook or signal check does not establish N. N requires a path with an explicit success value or unchanged propagation to a path already proven to return successfully.
- **Build capability.** An LSM-BPF result is changed from E to U if the Analyzer cannot establish that the target build enables the required BPF-LSM support. A valid hook does not guarantee enforcement when a BPF program cannot attach to it.
- **Artifact validity.** Before classification, the Analyzer verifies that `vmlinux` and `System.map` exist and are newer than the kernel source and `.config`. If the artifacts cannot be associated with the analyzed build, conclusive E and N results are downgraded to U.

The internal report preserves N and U separately. For LSM-BPF, the *Non-Enforceable* results reported in Section 5 consist of 138 N and 59 U syscalls; for Conventional SIGKILL, they consist of 257 N and 82 U syscalls. Because U is never counted as enforceable, the reported *Preemptively Enforceable* coverage is a lower bound on the coverage established by the analysis.

The Analyzer identifies unavailable syscalls by comparing syscall-table entries with symbols in `System.map`. Entries without an implementation in the target build are classified as N/A; this accounts for the 17 syscalls excluded in Section 5. Finally, the frontend caches ASTs, CFGs, and call relations by source and build configuration. When only a subset of syscall classifications must be updated, the Analyzer reuses unchanged intermediate results instead of reanalyzing the entire kernel.

C Syscall Skipping Path via ptrace_notify()

Listing 3 shows the simplified syscall-entry control flow used by SYGMA for preemptive blocking. When a Seccomp filter returns `SECCOMP_RET_TRACE`, `__seccomp_filter()` invokes `ptrace_event()`, which reaches `ptrace_notify()`. The SYGMA Policy Enforcer attached at `ptrace_notify()` evaluates the installed policy and, on a denial, marks `SIGKILL` as pending for the current task. After `ptrace_event()` returns, the Seccomp trace path evaluates `fatal_signal_pending(current)`. A pending fatal signal directs control to `skip`, which returns `-1` to the syscall-entry caller instead of allowing normal dispatch to continue.

This `-1` propagates through `syscall_enter_from_user_mode()` to `do_syscall_64()` as the syscall number `nr`. During normal execution, a nonnegative `nr` identifies the handler to be dispatched. When `nr` is `-1`, however, the x86-64 and x32 lookup paths do not dispatch the original handler, and the explicit `nr != -1` guard also prevents invocation of the fallback `__x64_sys_ni_syscall()` handler. Control then proceeds directly to `syscall_exit_to_user_mode()`, where the pending fatal signal is delivered. Thus, the policy-violating syscall never reaches its handler, and SYGMA enforces the denial before actual syscall execution rather than terminating the task only after the syscall has executed.

Listing 3: Seccomp in Linux 5.15 (x86-64)

```
__visible noinstr void do_syscall_64(struct pt_regs *regs
     , int nr)
{
    // Invokes __seccomp_filter.
    // Returns -1 if signal is pending.
    nr = syscall_enter_from_user_mode(regs, nr);

    // __x64_sys_ni_syscall is skipped due to nr != -1
    if (!do_syscall_x64(regs, nr) &&
        !do_syscall_x32(regs, nr) && nr != -1) {
        regs->ax = __x64_sys_ni_syscall(regs);
    }
    syscall_exit_to_user_mode(regs);
}

/* Based on Linux 5.15 x86-64 */
static int __seccomp_filter(int this_syscall, const
     struct seccomp_data *sd, const bool
     recheck_after_trace)
{
    switch (action) {
        case SECCOMP_RET_TRACE:
            // Triggers ptrace_notify() internally
            ptrace_event(PTRACE_EVENT_SECCOMP, data);

            // Skip if Enforcer sent a fatal signal
            // (e.g., SIGKILL)
            if (fatal_signal_pending(current))
                goto skip;

        return 0;
    }

skip:
    // Instructs the caller to skip the syscall
    return -1;
}
```

D Policy Activation on Online Boutique

This section presents the per-container Online Boutique results reported in Section 7.3. As shown in Figure 13, SYGMA activates the policy a median of 23.10–26.60ms before the first syscall across all containers. In contrast, Tetragon's activation gap varies substantially across containers, from 97.70ms for `redis-cart` to 831.80ms

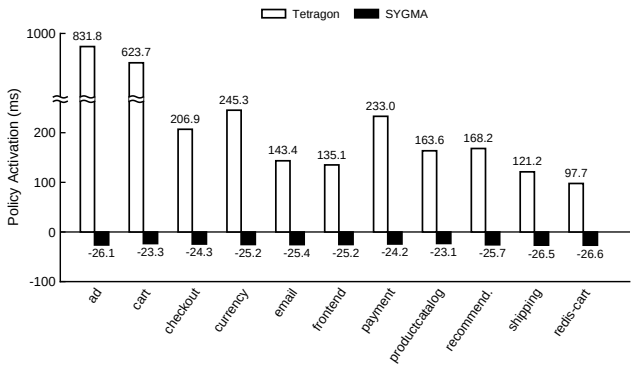


Figure 13: Policy activation times for the 11 containers in Online Boutique. Each value is the median of 10 trials relative to container entrypoint execution at time zero; negative and positive values indicate activation before and after the entrypoint, respectively.

for ad. Despite differences in container images and execution configurations, the ordering of SYGMA policy activation, first syscall execution, and Tetragon policy activation remains unchanged across all measurements for the 11 containers. This result shows that the effect of lifecycle synchronization observed in the main text remains consistent across Online Boutique, including its infrastructure containers, rather than being limited to a specific application container.