Santa: Language Agnostic Automated System Call Policy Learning for Cloud Microservices

Meghna Pancholi[1], Andreas Kellas[1], Kostis Kaffes[1], Steven M. Bellovin[1], Simha Sethumadhavan[1], and Vasileios P. Kemerlis[2]

[1] Columbia University, New York NY, USA
{meghna, kkaffes, smb}@cs.columbia.edu
{andreas.kellas, simha}@columbia.edu
[2] Brown University, Providence RI, USA
vpk@cs.brown.edu

Abstract. Attackers commonly abuse the system call API when performing attacks on cloud services, such as control-flow hijacking, privilege escalation, and container escape. We present SANTA: a paradigm for generating runtime-specific security policies and a technique to harden cloud-based applications from system call abuse. SANTA targets the microservices architecture, and automatically creates fine-grained system call policies via a runtime policy-learning framework. SANTA addresses many shortcomings of state-of-the-art techniques, such as the inability to generate tight and accurate filters for programs even if they are written in interpreted languages. Crucially, SANTA offers a solution without sacrificing performance, scalability, or engineering simplicity. In real-world microservices, our system generates system call filters that are on average 46.0% smaller (i.e., more stringent) than static, automated filtering approaches, and with runtime overheads 41.15% lower (on average) than microservices with traditional protection techniques applied.

Keywords: System call filtering · Attack surface reduction · Adaptive hardening · Filter learning · Microservices · eBPF

1 Introduction

Traditional and serverless microservices have become the preferred deployment choice due to their modularity, flexibility, and scalability in distributed cloud environments [48,17,67,33]. However, they introduce three security challenges: (1) *dynamism*—rapidly evolving architectures make it hard to keep security policies up to date [26]; (2) *heterogeneity*—microservices built by different teams in many languages on multiple platforms expand the attack surface [58]; and (3) *software bloat*—extensive libraries and APIs produce large container images with complex dependencies and hidden vulnerabilities [3,57]. These issues risk compromise and container-escape attacks [43], potentially endangering other tenants, while strict end-to-end latency requirements complicate mitigation [77]. To execute these attacks, adversaries invariably abuse the system call interface to escape the container or compromise the host.

Consequently, state-of-the-art efforts in securing microservices mitigate vulnerabilities in system calls by reimplementing, filtering, and removing them, or even removing the OS kernel entirely. However, such efforts often clash with performance goals. While user-space syscall emulators (e.g., gVisor [6]) and MicroVMs (e.g., Firecracker [2]) provide strong isolation boundaries, they introduce virtualization overheads that can violate strict latency requirements of production cloud applications. Similarly, microkernel designs require significant engineering effort to port applications [42,50]. Consequently, system call filtering (e.g., seccomp-BPF) [44,28,40,55,56,36,59] remains the most attractive defense mechanism for cloud-native environments. It offers a pragmatic balance, allowing administrators to restrict kernel access, preventing privilege escalation and lateral movement, without incurring the heavy performance penalty of hardware-level virtualization or the complexity of unikernels [42].

However, modern techniques for securing microservices with system call filtering have limitations that hamper their effectiveness and deployability. Traditional static analysis approaches [29,9,22,59] struggle with the heterogeneity of the microservice landscape. They often fail to generate accurate filters for interpreted languages (e.g., Python, Node.js) or complex frameworks that use dynamic class loading, leading to over-approximate policies that leave the attack surface unnecessarily wide. Conversely, dynamic analysis techniques rely on training runs that fail to cover all legitimate execution paths, leading to false positives that break application functionality in production [76]. This forces operators to choose between permissive policies that offer little security or restrictive policies that cause outages, rendering automated syscall filtering largely impractical for modern, rapidly evolving microservices.

We propose Santa, an approach for securing microservices [48,17,67] via novel dynamic syscall analysis that resolves the tension between strong security analysis and production latency. Santa iteratively builds runtime-specific system call policies per microservice using eBPF and a user-space controller to manage two versions of the application: a high-performance standard version and a hardened variant. The hardened variant serves as a flexible security oracle. Santa enables deploying high-overhead protection mechanisms (e.g., debug-time sanitizers, heavyweight binary rewriting, software-based fault isolation) that would otherwise be prohibitively slow for production use. Santa routes traffic to this oracle upon policy violations, providing the inspection capabilities of the hardened variant without constantly incurring its performance penalty.

Santa *automatically* builds a list of valid syscalls a microservice can invoke for each workload. The amortized performance of a Santa-protected system is similar to that of an unhardened system, nearly eliminating hardening overheads. Santa generates syscall sets 46.0% smaller (i.e., more stringent) than state-of-the-art static analysis tools. Additionally, Santa succeeds in creating syscall filters for interpreted languages with minimal effort, a challenge for static analysis-based techniques. Santa also provides defense-in-depth protection through incremental policy building, deny-list mechanisms, and multi-variant hardening.

2 Background

Microservice Architecture. Microservices have become a de-facto standard for web-based applications, with reported adoption rates as high as 74% [27], enabling teams to develop modular services independently and connect them via APIs [12]. Languages and runtimes like Python and Node.js [15] promote agile development, while Docker, Kubernetes, and cloud platforms simplify deployment and scaling. In multitenant environments, microservices must handle failures gracefully using *stateless* services with *retry semantics* [39] to ensure consistent results despite retries, crashes, or network congestion. Stateless frontends, isolated from stateful backends, are resilient against failures and handle idempotent requests until success [77,8]. Although microservices enable rapid development, they complicate security [17]. Frequent code and library updates make manual policy management infeasible [43]. Interpreted languages and external dependencies hinder automated code analysis [7], and co-located microservices on shared hosts increase the risk of container escapes and lateral attacks [14,60].
Stateless Microservices. Best practice separates stateless and stateful components, with stateless layers performing business logic and connecting frontends to databases and caches, similar to the three-tier architecture [61,17,4]. For computationally intensive workloads like machine learning or MapReduce-style processing, stateless services compute and pass values to stateful storage services [21,65]. Industry practice limits stateful components to data backends to optimize performance, cost, and manageability [49,32].

Stateless microservices are lightweight, allowing quick instantiation, migration, and autoscaling. Serverless platforms like AWS Lambda reduce costs by allocating resources only when needed. In contrast, stateful services incur higher costs due to persistent storage and availability, with scaling hindered by state consistency overhead. Isolating stateful components also reduces the risk of exposing sensitive data, enhancing security [13].
Attack Surface Reduction with System Call Filtering. The syscall interface allows user-space applications to request privileged services from the OS kernel. While the Linux kernel (v5.x) exposes approximately 350 syscalls [22], a typical microservice requires only a small subset [22,28,29,9,40,35]. However, attackers who gain arbitrary code execution can invoke *any* available syscall, violating the *principle of least privilege* [62]. This allows attackers to escalate privileges or exploit rarely used kernel paths [44,74]. Syscall filtering (e.g., seccomp-BPF) mitigates this by allowing administrators to restrict the kernel attack surface. By limiting a microservice to only necessary syscalls, operators can restrict attackers to developer-intended behaviors.

3 Motivation

System call filtering offers a balance between security and performance, but applying it to modern microservices creates a tension. Operators are currently forced to choose between heavy defenses that degrade performance, and lightweight filtering policies that are either inaccurate or brittle.

Isolation and Hardening Performance Costs. To mitigate the risks of shared kernels, some organizations turn to heavy isolation. User-space syscall emulators (e.g., gVisor [6]) and MicroVMs (e.g., Firecracker [2]) provide strong boundaries by intercepting syscalls or using hardware-level virtualization. However, these mechanisms violate the strict latency requirements of production cloud applications. The additional emulation layers or hypercall overheads add startup delays and runtime penalties that undermine containerization benefits. Similarly, microkernel designs [42,50] require significant engineering effort to port applications, limiting their adoption in heterogeneous environments.

Beyond isolation mechanisms, runtime hardening tools offer another approach to mitigating vulnerabilities. Memory safety tools like SoftBound [52] and AddressSanitizer (ASan) [63,66] effectively detect and prevent memory corruption, while race detection tools like ThreadSanitizer (TSan) [73,68,66] identify data races in concurrent applications. However, these tools introduce significant overheads ranging from 1.7x to 14.2x execution slowdowns and 4x to 9x increases in memory consumption, making them unsuitable for latency-critical network services [63,66,64,73]. The sidecar pattern colocates monitoring with microservices but incurs significant resource and performance overhead [78,34]. Consequently, low-overhead syscall filtering remains the most attractive defense if accurate policies can be generated.

Limitations of Existing Filtering. Generating accurate syscall filters remains an open challenge due to the complexity of modern microservices. Static analysis struggles with the heterogeneity of modern microservices, particularly interpreted languages and dynamic class loading [22,24,59]. To prevent crashes, tools must over-approximate their policies, negating security benefits. Conversely, dynamic analysis creates tighter filters but is brittle, as it is dependent on training runs that rarely cover all legitimate execution paths [9,40]. When valid but unseen workloads occur in production, strict filters cause availability outages, forcing operators to choose between security and uptime.

Operational Challenges. Microservices' dynamic nature complicates policy deployment and enforcement. In distributed clusters, microservices are scheduled on different nodes, requiring policies to be propagated beforehand [24]. Policies must remain intact across nodes while adapting to changing workloads, and different nodes may have different syscall requirements due to hardware, kernel versions, or runtime environment differences.

4 Threat Model

Goals. We consider a remote attacker whose goal is to compromise an application microservice. Upon successful exploitation, the attacker seeks to escape the container's isolation boundary, access host system resources, or laterally compromise other microservices co-located on the same physical machine.

Capabilities. The adversary interacts with microservices over exposed network interfaces by sending crafted inputs. They may exploit vulnerabilities in application code, language runtimes, or library dependencies to execute unauthorized

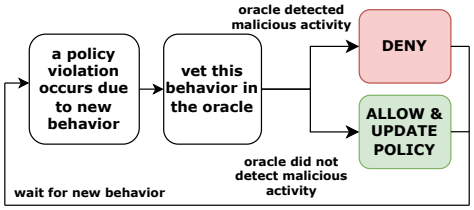

Fig. 1: SANTA supports iterative updates to a security policy. When new behavior is observed that does not conform with an existing security policy, it attempts to recreate it in a hardened variant to determine whether to update the policy.

code and issue arbitrary syscalls within the containerized microservice. We assume the attacker has only network access (no physical access), and we exclude side-channel [41,46,16,37] and fault-injection [70,51] attacks, focusing instead on behavior at the syscall interface. The attacker does not start with insider credentials or privileged access.

System Model and Attack Surface. Microservices are deployed as containerized applications across a distributed cloud cluster. Each runs in its own container, isolated from host resources, and communicates with other microservices or clients via network APIs. The primary attack surface is the syscall interface exposed by the runtime once remote code execution is achieved. The attacker may open arbitrary network connections, modify filesystem state, escalate privileges, or interact with kernel subsystems in unintended ways by abusing available syscalls. They exploit some vulnerability or misconfiguration to execute code in the container process [69], and aim to escape the container or compromise the application by executing arbitrary syscalls [10,1,44]. Our threat model aligns with prior work in this area [22,59,29,9,30,24].

5 Design and Implementation

We describe the design and implementation of SANTA, which learns syscall policies on the fly, including only observed syscalls and detecting malicious behavior when a new syscall is introduced. Fig. 1 illustrates the sequence of events when a service running with SANTA receives an input causing a policy violation.

5.1 Approach Overview

SANTA uses high-overhead, principled techniques to derive low-overhead, high-fidelity policies. By introducing a hardened variant to discern between policy violations caused by benign vs. malicious requests, we incrementally build tight, precise policies. The hardened variant is an instrumented service that alarms on malicious behavior, enabling SANTA to leverage heavyweight instrumentation's security benefits while maintaining production performance.

The key insight is that most production workloads follow predictable patterns, and the majority of syscalls are invoked during application initialization and common request code paths. By learning these patterns through selective hardening, SANTA creates policies that are more precise than static analysis approaches and more comprehensive than dynamic recording techniques. The hardened variant serves as a security oracle, validating whether new syscall invocations represent legitimate application behavior or exploitation attempts.

When a policy violation occurs for a specific input, we consult the hardened variant by re-executing the program with the current policy and input. If the hardened variant completes without detecting an exploit, we update the policy to *allow* the observed event and restart the original service with the new policy. Otherwise, we treat the input as an exploit attempt, *prevent* it, and raise an *alert*. Over time, repeated violations and re-executions incrementally build precise and tight policies tailored to the workload.

Correctness Requirements. For this approach to succeed, the program must be resilient to repeated re-executions from a known state. Network services, particularly microservices, are often designed to be stateless and support re-execution of idempotent requests (§2) [8,39]. This allows SANTA to stop a microservice for policy learning, let the request fail, and resend it without affecting the correctness of disjoint stateful services. We discuss limitations and compatibility with stateful services in Section 7. SANTA also does not require perfect deterministic replay of internal execution paths (e.g., thread interleaving). If a benign request triggers a system call in the production variant but, due to thread scheduling, does not trigger it in the hardened oracle, no policy violation occurs. The system call will eventually be observed and learned during subsequent oracle executions of similar requests.

For engineering simplicity, our system requires clients to resend requests until success, as our benchmarks do not implement automatic retries. This matches the behavior of idempotent microservice clients that retry for robustness [20]. Automatic retries could be added to the SANTA controller, but we leave this as future work. Real-world microservices are expected to tolerate crashes, migrations, or faults and use retries to achieve this [71], as evidenced by first-class retry support in service meshes like Istio [34].

5.2 System Call Policy Learning

We prototype SANTA on Kubernetes (K8s). The SANTA controller, deployed as a K8s `DaemonSet`, runs as a Pod on each node with a user-space component and kernel-space eBPF counterpart. These components manage policies and orchestrate microservices on every node, working together as shown in Figure 2.

When the eBPF program detects a syscall policy violation, it stops the program, logs the event, and sends it to a ring buffer polled by the user-space program. Upon detection, the user-space program uses the K8s API to kill the offending Pod and restart it as a hardened variant (§5.3). When the new hardened Pod is created, we update syscall policies based on its observed behavior.

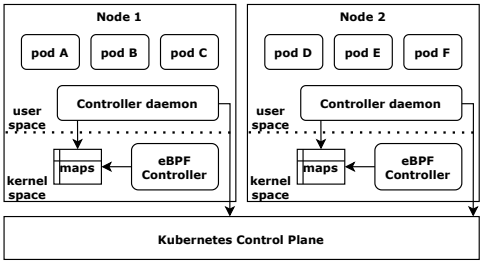

Fig. 2: The diagram shows the architecture of the SANTA system deployed on a cluster with two nodes—each node runs multiple K8s Pods. The SANTA controller daemon operates in user space, interfacing with the eBPF SANTA controller in kernel space, which utilizes BPF maps for managing data. The entire system is orchestrated by the K8s Control Plane, ensuring coordination across nodes.

The hardened variant runs *with the same syscall allow-list as the original service*. Given identical input, it should either (1) exit due to the same syscall violation, or (2) exit from an ASan/TSan abort. Case (1) implies the code path to the syscall did *not* cause a sanitizer violation, allowing the offending syscall to be safely added to the allow-list. The controller then updates the policy and restarts the original service. Case (2) indicates a sanitizer violation occurred before the new syscall, deeming the input unsafe. After a timeout, the hardened variant is killed, and the original Pod restarts with the policy learned during its execution. We set the timeout to 30 seconds to balance killing and restarting overhead with running the hardened variant too long. This policy is maintained by the SANTA controller regardless of restarts.

The user space component communicates with the K8s control plane and manages the execution of hardened or unhardened microservices. It runs as a privileged Pod to install the eBPF program in the host's kernel and uses the K8s API for Pod orchestration.

The SANTA eBPF program, a C program using the `libbpf` framework, builds and enforces system call policies. Unlike `seccomp-BPF` (cBPF) that often stores policies as files [24], our custom eBPF program stores policies for each microservice in an array in kernel space. The `sys_enter` eBPF tracepoint intercepts all system calls and inspects the `task_struct`. To enforce policies, the eBPF program maps containers to microservices at system call invocation. Since `task_struct` has only kernel-level identifiers, we enrich the eBPF program with user space information from Pod creation. Using eBPF hooks at `sched_process_exec` and `sched_process_fork`, we intercept container runtime commands to link the `cgroup` name to the microservice/Pod name, allowing identification of the container ID invoking a system call and policy comparison from a container's inception.

To minimize overhead, the hardened variant's exploit detection is used only upon policy violations. This ensures the system primarily runs the optimized container image, opportunistically leveraging exploit detection. (We discuss the runtime performance of SANTA in Section 6.2.)

Crucially, SANTA begins monitoring only after container initialization completes. Unlike static policies that must permit all initialization syscalls, SANTA ignores these one-time events, preventing the policy from being artificially loosened by routines never used during steady-state execution.

5.3 Hardened Variant Considerations

Designing an effective hardened variant involves several considerations: (1) ability to detect abuse—the hardened variant needs mechanisms to detect exploitation attempts; (2) behavior alignment—security instrumentation must not alter the microservice's core behavior; (3) support for retrying requests—the hardened variant must retry the same request without adverse side effects; and (4) minimization of policy-loosening actions—if the hardened variant uses significantly more syscalls than the original, the resulting policy could be too permissive.

Our prototype uses ASan/TSan for demonstration, but is not limited to them. ASan detects runtime memory-safety violations (≈70% of CVEs annually [72]); TSan detects data races exploitable for control-flow hijacking or privilege escalation [68]. We compile with ASan/TSan using Clang. These variants satisfy the above properties; we discuss their effectiveness in Section 6.3.

When a hardened variant (oracle) requires certain new system calls that are not needed by the production service, SANTA can maintain a separate *hardened-only* syscall policy. This policy should include only system calls that are: (1) essential for the hardened variant's runtime protection mechanisms (e.g., thread synchronization syscalls for race detectors); (2) not security-sensitive in the context of the hardened variant's execution environment; and (3) excluded from the production service policy to maintain a tighter attack surface. The eBPF program can enforce different policies based on whether a container is running the hardened variant or the production service. For our ASan/TSan hardened variants, this separation was not necessary, as the additional syscalls introduced by these sanitizers are minimal and pose low security risk (see Section 6.3).

6 Evaluation

6.1 Testbed

We analyze SANTA by executing five diverse microservice benchmarks representative of real-world applications, featuring multiple languages, coding patterns, libraries, and multithreading. These benchmarks span simple web services to complex multi-tier applications with databases, caches, and external APIs. They mirror the complexity of actual deployments, including dependency management, concurrent requests, and coordination across services. Running on

Amazon EKS, they are suitable for evaluating both security and performance under realistic conditions. While we focus on traditional microservices, SANTA is adaptable to serverless environments with minor engineering changes.

Cluster Setup. We run our experiments on Amazon Elastic K8s Service using `m5.large` instances with two vCPUs and 8 GB of RAM each. This setup approximates realistic cloud deployments and is well-suited for evaluating the performance and security implications of running SANTA.

Benchmarks. To evaluate SANTA in a realistic microservice environment, we use applications from DeathStarBench [26], Google's Online Boutique, and Istio's Bookinfo. These benchmarks model widely used web applications like media streaming, social networks, and hotel reservation systems. They depend on common technologies (Memcached, Redis, Thrift, MongoDB, Nginx) and use a variety of languages and communication libraries.

Load Generation. We use `locust` [47] to generate load, simulating multiple simultaneous users. Wait time follows an exponential distribution with mean 0.02 seconds, and we vary users from 1 to 500. This drives requests across tiers and forces services through many SANTA policy-learning iterations, making the workload suitable for both performance and security evaluation. The 500-user load is intentionally heavy for our setup, highlighting SANTA's overheads under saturated services and long queues.

Application Correctness. While evaluating SANTA, we ensure that application state remains correct. We verify that each request is eventually processed successfully, re-issuing failed requests until they succeed. This preserves global application state even when policy learning restarts cause individual failures. We also ensure that applications do not crash or hang by monitoring application and system logs throughout the experiments.

6.2 Performance

SANTA aims to provide hardened security with amortized performance comparable to an unhardened system. We compare applications running under SANTA to two baselines: an unhardened configuration with no additional exploit mitigations and a fully hardened configuration using ASan/TSan (see Table 2 for hardened variants per microservice). This three-way comparison highlights the security and performance tradeoffs between a completely instrumented system, a defenseless system, and a SANTA-hardened system.

All SANTA trials begin by executing the test application with an empty system call allow-list while `locust` generates workloads. Some syscalls are iteratively learned before the application receives its first input due to syscalls required to initialize and start running the application. We measure end-to-end latency as the elapsed time between a client issuing a request and receiving its response, capturing the full impact of SANTA's policy learning overhead, including hardened variant execution, policy updates, and service restarts.

In all experiments, the first request sent to the application resulted in slow response times since handling the first request required learning many new system calls. Subsequent requests tended not to invoke many new system calls, and were

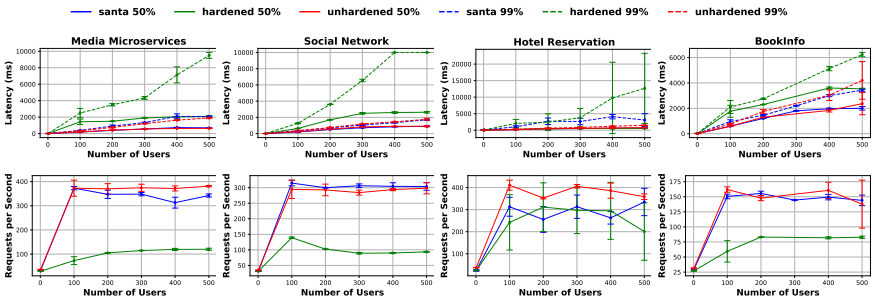

Fig. 3: The first row of graphs are a comparison of 50th percentile and 99th percentile latency of requests sent to an unhardened service, a SANTA service learning system call policies, and a hardened service. The second row of graphs compare the achieved requests per second in ten minutes of execution with SANTA, hardened, and unhardened systems.

therefore quickly processed. Fig. 3 shows the distribution of end-to-end latencies for different loads in the first second and 100 seconds of application execution (results from Google's Online Boutique are available in the appendix). We learn the majority of system calls during the initial phases of application execution. We also see more outliers in the first phase due to the overheads incurred from processing requests with the variant during policy learning. Across these experiments, the average requests per second of a SANTA app is 97.96% greater than that of a hardened application. The average requests per second for a SANTA app is 7.48% less than that of an unhardened application.

Minimizing Tail Latency. Policy violations cause degradation as SANTA substitutes a microservice with its hardened variant, incurring K8s restart and hardening overhead. This can be minimized by reducing the hardened variant timeout and through pre-training.

The current 30-second hardened variant timeout parameter represents a direct trade-off aimed to minimize tail latency: a longer timeout increases the time we spend using the high-overhead variant, while a shorter timeout minimizes the number of system calls learned per restart, potentially leading to more frequent restarts. Since restarts are slow and often contribute to tail latency, we found that the 30-second timeout provided a good balance—it allows us to learn many system calls per restart while keeping the time spent in the hardened variant manageable. Since many system calls are learned at the beginning of execution, we set this timeout value to be large initially, but it could be reduced after the first initialization window to incur smaller performance penalties.

Pre-training an application in SANTA on known-safe inputs to create initial allow-lists in advance reduces frequent SANTA restarts at runtime. This allows the application to run with SANTA's tight policies, immune to crashes if low-probability system calls are invoked. This approach offers stronger security and performance than dynamic training for system call filter generation [9,40], as

Table 1: SANTA provides strong security benefits in comparison to previous works that rely on generating a static system call filter. We define static system call filtering tools as those that generate a filter prior to execution and only enforce that filter throughout application execution [22,28,29]. Our tight and incremental filter building process results in a smaller attack surface and restricts attacker capabilities when a service is under attack.

Attacker Capabilities in SANTA vs. Static System Call Filtering

	New System Calls Executed	No New System Calls Executed
Hardened Variant-Detectable Vulnerability	**(1)** SANTA always detects the attack and stops execution, but static syscall filtering can only do so if the filter is adequately tight.	**(2)** The attacker is limited to using the existing syscall filter. The filter is tighter with SANTA since it only adds previously-seen syscalls.
Hardened Variant-Undetectable Vulnerability	**(4)** When SANTA is supplemented with filters derived by static analysis, its security is equal to that of static syscall filtering.	**(3)** The attacker is limited to using the existing syscall filter. The filter is tighter with SANTA since it only adds previously seen syscalls.

SANTA continues runtime system call learning. If a service's input set changes, tail latency is unavoidable, but SANTA automatically restarts and updates policies, unlike techniques requiring manual intervention for false positives. With multiple microservice replicas, a simple SANTA extension could coordinate to ensure one replica always processes requests.

6.3 Security

We consider SANTA's security features and argue it provides protection given reasonable threat model assumptions.

Attack Categories Given a remote attacker (§4), we classify attack scenarios by whether the attacker invokes new syscalls and whether the exploit is detectable by the hardened variant. The four attack categories in Table 1 summarize the combinations.

Category **(1)** attacks (hardened-variant-detectable exploits that execute a policy-violating syscall) are detected and prevented by SANTA, which offers its strongest guarantees. When the application container violates the syscall policy due to offending input, the hardened variant detects the exploit trigger, prevents any policy updates, and raises an alert.

Category **(2)** and **(3)** attacks (exploits that do not execute a policy-violating system call) are handled similarly by SANTA. In these *mimicry attack* scenarios, SANTA's benefits align with the Principle of Least Privilege (PoLP) [62]. This limitation applies to all system call filtering; however, SANTA mitigates it more effectively than static analysis by learning a minimal subset for the runtime workload rather than a static over-approximation. The policy is never violated so the hardened variant is never consulted, but the attacker remains restricted to previously-observed syscalls. While prior work builds immutable full allow-lists, SANTA adds syscalls conservatively as observed, yielding a smaller allow-list.

Category **(4)** attacks (hardened variant-undetectable exploit that executes a policy-violating system call) raise a natural concern: a *policy-loosening* attack in which an adversary first sends purely benign requests that trigger new, legitimate-looking system calls (no malicious payload) so that SANTA adds them to the allow-list, then reuses those newly allowed syscalls in a second phase to pursue their goals. Even in this scenario, SANTA imposes concrete barriers to invoking arbitrary system calls without detection.

Barriers to Arbitrary-syscall Use. The allowed set can only grow to include *developer-intended* syscalls: i.e., those with a valid execution path in the microservice binary. While common implementations of the C standard library (`libc`) expose both the generic `syscall()` interface and wrapper functions for nearly all kernel syscalls, the mere presence of these interfaces does not permit arbitrary policy expansion. Invoking unused syscalls requires redirecting execution or manipulating program state through control-flow hijacking [69], data-oriented programming [45], or data-only attacks [38]. Such techniques inherently depend on memory-corruption—and often memory-disclosure—vulnerabilities in the process. Our architecture expects these vulnerabilities to be detected by the hardened variant (serving as a security oracle). Our prototype leverages ASan/-TSan for this purpose and is therefore constrained by the known limitations of these sanitizers (e.g., certain memory-safety violations may evade detection) [75]. This constraint reflects the capabilities/limitations of the underlying sanitization mechanisms rather than a fundamental weakness in the design of SANTA. Moreover, SANTA consults the hardened variant on every policy violation and records each such event. Any attempt to systematically broaden the allow-list toward "all `libc`-reachable" syscalls would trigger repeated violations, restarts, and policy revisions, which are both observable and operationally costly.

Mitigations and Worst-case Scenarios. We recommend: (1) choosing a hardened variant suited to the application and its vulnerability profile; and (2) maintaining a deny-list derived from over-approximate static analysis, the default Docker `seccomp-BPF` filter [28], or static syscall-filtering tools [22,59] so high-risk syscalls are never included in the SANTA policy. When SANTA is supplemented with such filters, its guarantees match those of static syscall filtering. As we show in Section 6.3 (RQ1, RQ2), SANTA policies are materially tighter than `SysPart` and `sysfilter` (≈46–49% fewer syscalls, and we exclude CVE-associated syscalls that those tools allow). Thus, even under policy loosening, the reachable set that an attacker could plausibly induce remains smaller than static over-approximations, and the deny-list blocks the most dangerous syscalls. In practice, SANTA thus offers security that is at least equal to, and often stronger than, static-analysis-based filtering.

SANTA Policies In addition to the qualitative analysis, we evaluate the quality of the system call policies created by SANTA. We ask: **(RQ1)** how many fewer system calls does SANTA allow than over-approximate static techniques; **(RQ2)** which system calls do static techniques allow that SANTA does not;

Table 2: Size of benchmark policies: SANTA- vs. `SysPart`- vs. `sysfilter`-generated vs. generated by recording syscalls. (*)/(-) = hardened by ASan/TSan.

Microservice	Santa	SysPart	Sysfilter	Rec.
Social Network				
ComposePost*	47	73	77	40
HomeTimeline*	40	73	77	26
Media*	38	73	77	25
PostStorage*	49	82	88	50
SocialGraph*	55	82	88	49
Text*	47	73	77	33
UniqueId*	39	73	77	39
UrlShorten*	55	82	88	49
UserMention*	50	82	88	48
User*	54	82	88	49
UserTimeline*	50	82	88	50
Media Microservices				
ComposeReview*	44	73	79	41
CastInfo*	55	82	84	49
MovieId*	46	82	84	45
MovieInfo*	56	82	84	46
MovieReview*	50	83	84	46
Page*	37	68	74	30
Plot*	56	82	84	49
Rating*	44	70	74	41
ReviewStorage*	46	82	84	46
Text*	43	68	74	34
UniqueId*	39	68	74	28
UserReview*	55	83	84	46
User*	49	82	84	49

Microservice	Santa	SysPart	Sysfilter	Rec.
Hotel Reservation				
Reservation−	39	-	-	38
Profile−	47	-	-	38
Geo−	38	-	-	38
Frontend−	38	-	-	31
Recommend−	44	-	-	31
User−	35	-	-	35
Rate−	47	-	-	43
Search−	53	-	-	38
Bookinfo				
productpage*	66	157	173	55
details*	66	129	140	43
reviews*	86	-	-	82
ratings*	46	137	137	39
Online Boutique				
productcatalog−	40	-	-	26
currency*	57	137	137	53
payment*	55	137	137	53
shipping−	37	-	-	18
email*	60	157	173	56
checkout−	45	-	-	42
recommendation*	58	157	173	55
ad*	73	-	-	69
frontend−	47	-	-	30

(RQ3) which system calls does SANTA allow that static techniques do not; and **(RQ4)** which system calls are introduced solely by SANTA's instrumentation.

To compare policies, we use the state-of-the-art static system call filtering tools `SysPart` [59] and `sysfilter` [22]. `SysPart` additionally isolates the set of system calls used for "initialization" from the set to achieve temporal system call filtering. We evaluated SANTA by running benchmarks under realistic workloads, and used an eBPF program to record baseline system calls from a K8s Pod to help identify syscalls introduced by sanitizers or the runtime. Table 2 summarizes these results (dashes denote policies that the tools could not generate). We excluded Go, Java, and C# microservices where `SysPart` and `sysfilter` were unable to extract system calls or, in the case of C#, where SANTA was not executed with a hardened variant.

RQ1. For all evaluated programs, `SysPart`- and `sysfilter`-created policies were at least as large as those created by SANTA. On average, the SANTA generated system call sets were 48.8% smaller than those generated by `sysfilter` and 46.0% smaller than those generated by `SysPart`, meaning that an attacker has access to approximately 48.8% or 46.0% fewer system calls when exploiting a system defended by SANTA compared to one defended by `sysfilter` or `SysPart`. Static approaches ignore the fact that inputs modulate program behavior at runtime and estimate the system call behavior solely based on static program code, thus over-approximating the set of allowed system calls. In contrast, SANTA *tailors* the syscall policies to only observed inputs, showing reduced attack surface and tighter PoLP enforcement.

RQ2. Not all system calls are equal in terms of security. Attackers require only a specific subset of system calls unique to their post-exploitation behavior. We analyzed the policies generated for two services: the `ComposePost` service from the Social Network application and the `ProductPage` service from the Bookinfo application. For both services, SANTA created smaller system call policies than `SysPart` and `sysfilter`. The SANTA policies for `ComposePost` and `ProductPage` were 26 and 91 system calls smaller than the `SysPart` filters respectively. Many of these extraneous system calls were innocuous, like `getrandom` and `time`. Others, however, may provide opportunities to compromise container isolation, with numerous previous CVEs reported. For example, `sysfilter` allowed both services access to `recv` (CVE-2016-10229), `recvfrom` (CVE-2015-2686), `dup` (CVE-2016-3750), `sigreturn` (CVE-2019-13648), and `setgroups` (CVE-2023-6507), while SANTA did not. Similarly, `SysPart` allowed both services access to `mremap` (CVE-2024-53111), `rt_sigreturn` (CVE-2024-35873), and `chmod` (CVE-2022-0492), while SANTA did not. A full table of system calls allowed by each system, along with a mapping to associated Linux kernel CVEs, is available in the appendix.

RQ3. Some system calls are allowed by SANTA and the syscall recording profile but are not permitted by the `sysfilter` or `SysPart` policy. This discrepancy arises from the difference between generating a policy solely from a binary and generating a policy from both the binary and its infrastructure, or from dynamically loaded libraries that are tedious to include in static analysis. When creating a SANTA policy for a containerized application, it is essential to include system calls invoked within the container that may not be required by just the binary. To effectively limit all system calls invoked within a container's `cgroup` namespace, policies must also encompass the system calls invoked by the cloud infrastructure within that namespace. These system calls are essential for deploying the microservice in a cloud setting.

RQ4. To determine which syscalls were introduced by SANTA, we compare the policies generated by SANTA and the ones seen in the syscall recording profile. These syscalls come from the instrumentation performed by the sanitizers. The syscalls we most often observed only in the C/C++ SANTA policy were `readlink`, `getrlimit`, and `sched_getaffinity`. The Go microservices included: `membarrier`, `fgetxattr`, `getegid32`, `gettid`, `fcntl`, `clone`, `rt_sigaction`, `rt_sigprocmask`, `sigaltstack`, `nanosleep`, and `madvise`, all related to thread synchronization and attributed to the TSan runtime.

Of these system calls introduced by SANTA, most incur a low risk in the context of privilege escalation. The one that is most concerning is `clone` since it could be abused to spawn new execution threads. The Go microservices can be protected by dropping all unnecessary Linux capabilities in K8s, particularly removing access to `CAP_SYS_ADMIN`. For hardened variants that introduce more security-sensitive system calls, SANTA can maintain a separate hardened-only syscall policy (see Section 5.3). Since SANTA is powered by a custom eBPF program, it can easily be modified to maintain separate policies if future hardened variants require more security-sensitive syscalls that should be excluded from the production service policy.

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

6.4 Case Study: CVE-2013-2028 (Nginx)

To demonstrate SANTA's real-world effectiveness, we case-study its protection of Nginx (v1.4.0) against CVE-2013-2028 [53], a stack buffer overflow vulnerability allowing arbitrary code execution. We use SANTA to deploy and protect an instance of Nginx v1.4.0, initialize a policy by browsing to the web server, and then use an existing exploit [18] to attack the service. The exploit abuses the stack buffer overflow to leak the stack canary, use a ROP [25] chain to change memory segment permissions, and execute a reverse shell. SANTA *successfully detected and prevented payload execution.* The exploit's initial attempt to leak the stack canary caused a worker process to attempt the `tgkill` system call, which was not in the learned policy. SANTA identified this disallowed system call and launched the hardened Nginx container. When the attacker retried the exploit, ASan in the hardened service detected the memory corruption attempt and thwarted the exploit.

7 Discussion

SANTA vs. Permanent Hardening. SANTA only checks inputs with sanitizers when they trigger new syscalls. If a malicious input induces memory corruption without invoking new syscalls, the SANTA production service could be compromised, whereas a permanently hardened service would deter it. However, real-world malicious payloads often invoke new syscalls [29]. Since permanently hardened services incur 1.7x–14x slowdowns and 4x–9x memory overheads, SANTA offers much of their security benefit without persistent costs. Further, while tools like ASan are often viewed as debugging aids, recent work demonstrates they effectively approximate Inline Reference Monitors (IRMs), providing strong security guarantees against memory corruption with low false negative rates [66].
Cluster Architecture Security. To deploy SANTA securely within a cluster, we address shortcomings of existing syscall restriction methods. Current solutions distribute policy files to nodes or rely on the permissive `seccomp-BPF` filter. Managing these files is complex: policies must be correctly placed, updated, and migrated when pods reschedule, and ensuring their integrity is challenging [24]. SANTA simplifies this by building and storing policies directly in the node's kernel, reducing the attack surface and eliminating manual file management. While the SANTA controller runs as a privileged Pod, this does not inherently expose the cluster to new vulnerabilities. Like other essential K8s components, it interacts securely with the Control Plane and is not exposed to user input.
Denial-of-Service. SANTA's design assumes microservices are idempotent and can retry requests. An attacker who continuously triggers syscall policy violations can force restarts into the hardened variant, causing denial-of-service

(DoS). This temporarily degrades performance (exploit detection and orchestration overhead) and consumes resources like traditional DoS, but does not compromise integrity or confidentiality. In simulation, sending malicious requests to the original service and benign requests to the hardened variant added no malicious syscalls to the policy, leaving the attack incomplete. When a benign request triggered the hardened variant and a malicious request was then sent to it, the hardened variant still detected the exploit independently. SANTA requires successful re-execution under the hardened variant to incorporate new behavior; malicious requests alone cannot subvert this. This DoS risk is not unique to SANTA and does not undermine its security guarantees. Mitigations include rate-limiting, replica rotation, or anomaly detection to throttle malicious clients [11]. Because SANTA's security model does not rely on availability for enforcement, even under repeated restarts or delayed responses the attacker cannot bypass policy learning or inject unauthorized syscalls.

Compatibility with Stateful Microservices. SANTA targets stateless microservices that interact with stateful backends [39]. We apply SANTA to all but database and caching services. Stateful services are accessed via SANTA - protected services, transitively enhancing their security. However, these benefits do not hold if the attacker crafts an attack that does not exploit any of the stateless services, but directly targets the stateful service. With future work to enable resilience and accuracy for stateful components, SANTA can be applied to stateful services without correctness issues during policy learning.

Reusing Policies Across Deployments. In a distributed cluster, workloads frequently migrate across nodes. SANTA allows for two synchronization strategies: (1) *Fresh Initialization*, where each new replica starts with an empty policy—this is what we recommend and implement, as it offers the highest security by ensuring the filter is strictly limited to the specific node's hardware and kernel version; and (2) *Policy Transfer*, where replicas can inherit a policy from an existing replica—this significantly reduces tail latency by avoiding redundant learning phases, but introduces a risk where a slightly over-approximate policy might be used if the workloads between replicas differ. With several simultaneous replicas, a policy could also be initialized via a consensus of system calls learned across replicas, balancing rapid convergence with stringent filtering.

Limitations and Opportunities. SANTA's security hinges on the strength of the hardened variant to detect malicious behavior, and, consequently, the availability of such a hardened variant. ASan and TSan are designed for C/C++ programs and rely on specific compiler and runtime support not available in the .NET runtime used by C#—C# runs in a managed environment with built-in memory safety features. However, in the presence of other threats, we would need additional runtime safety instrumentation to use as hardened variants. SANTA is illustrated with sanitizers, but is not dependent on them—they can be replaced entirely or supplemented by other mitigation schemes. SANTA's approach even encourages the development of additional high-overhead security mechanisms that would otherwise be impractical or expensive in production settings since it only selectively uses the protection mechanism.

8 Related Work

Static System Call Filtering. Related work struggles with microservices because arbitrary cloud deployments rely on containerized applications with complex runtimes, interpreters, and dynamic loading that complicate static analysis. Static system call filtering runs on raw binaries without executing the program. `sysfilter` [22] is a binary analysis-based framework that synthesizes and applies syscall filters to applications. Like SANTA, it determines developer-intended program behavior, but enforces it via `seccomp-BPF`, using static call-graph analyses. Despite using techniques to prune the graph to avoid unreachable syscalls, its filters are over-approximated by design, as static analysis cannot account for runtime behavior variations, dynamic library loading, or language-specific runtime system calls that are common in microservice environments.

Abhaya [54] applies a similar strategy to source code, targeting `seccomp-BPF` and OpenBSD's `pledge`. Bastion [35] adds context-aware static filtering through argument analysis and hardware-assisted shadow stacks, but requires custom compiler passes that increase deployment complexity. Saphire [7] improves policy generation for PHP via source analysis [19], but is limited to PHP and would require substantial re-engineering for other languages or even new PHP versions. Confine [28] reduces container attack surface via static analysis of container images, but cannot easily adapt to runtime variation.

In contrast, SANTA's approach avoids static over-approximation by learning policies dynamically, enabling tighter security without sacrificing agility or compatibility in heterogeneous, rapidly evolving deployments.

Dynamic System Call Filtering. Dynamic filtering executes the target application with various inputs and derives syscall policies from observed behavior. While these approaches can capture runtime-specific behavior, they typically separate training and deployment phases, making it hard to react when new behaviors emerge post-training. ZenIDS [31] generates policies for PHP with online training but cannot adapt once in production. Systrace [55] also builds policies via dynamic tracing but relies on a user-space daemon for enforcement, adding operational overhead. SANTA integrates learning into production, eliminating separate training. Instead of terminating on violations, SANTA treats new syscalls as refinement opportunities. Hardened variants validate new behavior; services pause only briefly to refine policies.

Fine-grained System Call Filtering. Fine-grained filtering focuses on context-aware syscall policies. `SysXCHG` [24] adapts policies across program phases, solving Seccomp's inheritance problem by switching on `execve[at]`. Temporal filtering [29,59] builds separate policies for initialization and steady-state phases, but may require manual annotations or prior behavior knowledge, limiting scalability. Blair et al. [5] propose stateful policies bridging static and dynamic filtering. Their μPolicyCraft framework micro-executes binaries to synthesize stateful, configuration-specific policies. While precise, it incurs non-trivial overhead and depends on symbolic execution and specialized micro-execution environments. SANTA avoids such complexity by learning policies dynamically from production workloads, achieving tight policies without extensive offline analysis.

SANTA offers a flexible syscall filtering framework that can be enhanced with temporal filtering or fine-grained, context-aware argument filtering. `SysXCHG`'s policy switching at `execve[at]` can be integrated with SANTA for phase-specific policies, while a sliding window achieves temporal filtering. What sets SANTA apart is delivering a runtime policy-learning framework tailored to microservices, enabling continuous policy refinement without disrupting availability. Unlike static analysis, which requires language-specific customization per environment or codebase, SANTA provides a language-agnostic solution that eliminates the cost and complexity of maintaining bespoke toolchains.

Hardened Runtime Environments. Cloud-hardened runtimes limit OS access, primarily via syscalls [6,2,42]. gVisor [6] intercepts syscalls in a user-space "kernel," isolating containers from the host but adding overhead. Unikernels like Unikraft [42] bundle OS and app in single-address-space images for strong isolation with lower overhead. Shard [1] limits kernel function access for fine-grained control but shares the overhead and compiler dependencies of similar schemes. Firecracker [2] provides KVM-based microVMs with VM-level isolation at higher resource cost than containers; it supports `seccomp-BPF` [23], compatible with SANTA. MicroVMs offer stronger isolation but slower startup and less mature auto-scaling than containers. SANTA and containers will remain relevant for many legacy and existing workloads.

9 Conclusion

We presented SANTA, a framework for automatically creating system call policies without performance sacrifices. SANTA's core idea is to alternate execution between two application variants: one hardened for security and one optimized for performance. SANTA learns execution properties from the hardened version and crafts policies enforced on the optimized counterpart, taking a pragmatic stance regarding the trade-off between security and performance. We demonstrated SANTA's merit and applicability in three ways: syscall allow-lists created by SANTA are significantly more precise than statically-generated policies; SANTA's compatibility with modern software engineering practices, like the use of microservices crafted using popular interpreted languages; and that the idea can be easily implemented and deployed in the modern cloud landscape. In our community there has been significant work on hardening applications, often coming with nontrivial performance and energy overheads. In reality, users care about performance, forcing them to make a hard choice between efficiency and security. By using a hardened variant strategically to learn security properties and then enforcing them with lightweight mechanisms, SANTA obviates the need to choose between performance and security. With time, we expect properties and policies beyond system call filtering to use the SANTA approach to hardening.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the National Science Foundation (NSF), through awards CNS-2238467, CNS-2212479, and DGE-2036197, and the Office of Naval Research (ONR), through award N00014-20-1-2746. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, ONR, or NSF.

Availability. The prototype implementation of Santa is available at: https://github.com/meghna-pancholi/santa-ebpf.git

References

1. Abubakar, M., Ahmad, A., Fonseca, P., Xu, D.: SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In: USENIX Security Symposium (SEC). pp. 2435–2452 (2021)
2. Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., Popa, D.M.: Firecracker: Lightweight virtualization for serverless applications. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 419–434 (2020)
3. Agadakos, I., Jin, D., Williams-King, D., Kemerlis, V.P., Portokalidis, G.: Nibbler: Debloating Binary Shared Libraries. In: Annual Computer Security Applications Conference (ACSAC). pp. 70–83 (2019)
4. Amazon Web Services: AWS Serverless Multi-Tier Architectures with Amazon API Gateway and AWS Lambda. https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/three-tier-architecture-overview.html (2021), accessed: 2024-10-23
5. Blair, W., Araujo, F., Taylor, T., Jang, J.: Automated Synthesis of Effect Graph Policies for Microservice-Aware Stateful System Call Specialization. In: IEEE Symposium on Security and Privacy (S&P). pp. 4554–4572 (2024)
6. Brewer, E.: gVisor: Protecting GKE and serverless users in the real world. https://cloud.google.com/blog/products/containers-kubernetes/how-gvisor-protects-google-cloud-services-from-cve-2020-14386
7. Bulekov, A., Jahanshahi, R., Egele, M.: Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists. In: USENIX Security Symposium (SEC). pp. 2881–2898 (2021)
8. Burckhardt, S., Chandramouli, B., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C.S., Zhu, X.: Netherite: Efficient Execution of Serverless Workflows. Proceedings of the VLDB Endowment **15**(8), 1591–1604 (2022)
9. Canella, C., Werner, M., Gruss, D., Schwarz, M.: Automating Seccomp Filter Generation for Linux Applications. In: Cloud Computing Security Workshop (CCSW). pp. 139–151 (2021)
10. Canister: Container Escapes: An Exercise in Practical Container Escapology. https://capsule8.com/blog/practical-container-escape-exercise/
11. Carl, G., Kesidis, G., Brooks, R.R., Rai, S.: Denial-of-Service Attack-Detection Techniques. IEEE Internet Computing **10**(1), 82–89 (2006)
12. Center, A.A.: N-tier architecture style. https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier
13. Chandramouli, R.: Security Strategies for Microservices-based Application Systems. NIST Special Publication **800**(204), 1–50 (2019)
14. Chierici, S.: CVE-2022-0492: Privilege escalation vulnerability causing container escape. https://sysdig.com/blog/detecting-mitigating-cve-2022-0492-sysdig/
15. Christou, G., Ntousakis, G., Lahtinen, E., Ioannidis, S., Kemerlis, V.P., Vasilakis, N.: BinWrap: Hybrid Protection against Native Node.js Add-ons. In: ACM ASIA Conference on Computer and Communications Security (ASIA CCS). pp. 429–442 (2023)

16. Christou, N., Gaidis, A.J., Atlidakis, V., Kemerlis, V.P.: Eclipse: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies. In: ACM Conference on Computer and Communications Security (CCS). pp. 3913–3927 (2024)
17. Cockcroft, A.: Evolution of Microservices – Craft Conference. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference
18. danghvu: danghvu/nginx-1.4.0. GitHub, https://github.com/danghvu/nginx-1.4.0
19. David, Y., Christou, N., Kellas, A.D., Kemerlis, V.P., Yang, J.: Quack: Hindering Deserialization Attacks via Static Duck Typing. In: Network and Distributed System Security Symposium (NDSS) (2024)
20. Dean, J., Barroso, L.A.: The Tail at Scale. Communications of the ACM **56**(2), 74–80 (2013)
21. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM **51**(1), 107–113 (2008)
22. DeMarinis, N., Williams-King, K., Jin, D., Fonseca, R., Kemerlis, V.P.: sysfilter: Automated System Call Filtering for Commodity Software. In: International Symposium on Research in Attacks, Intrusions and Defenses (RAID). pp. 459–474 (2020)
23. Firecracker Maintainers: Seccomp in Firecracker. https://github.com/firecracker-microvm/firecracker/blob/main/docs/seccomp.md (2024)
24. Gaidis, A.J., Atlidakis, V., Kemerlis, V.P.: SysXCHG: Refining Privilege with Adaptive System Call Filters. In: ACM Conference on Computer and Communications Security (CCS). pp. 1964–1978 (2023)
25. Gaidis, A.J., Moreira, J., Sun, K., Milburn, A., Atlidakis, V., Kemerlis, V.P.: FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In: International Symposium on Research in Attacks, Intrusions and Defenses (RAID). pp. 527–546 (2023)
26. Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., Lin, R., Liu, Z., Padilla, J., Delimitrou, C.: An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In: ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 3—18 (2019)
27. Gartner Peer Community: Microservices Architecture: Have Engineering Organizations Found Success? (2023), https://www.gartner.com/peer-community/oneminuteinsights/omi-microservices-architecture-have-engineering-organizations-found-success-u6b
28. Ghavamnia, S., Palit, T., Benameur, A., Polychronakis, M.: Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In: International Symposium on Research in Attacks, Intrusions and Defenses (RAID). pp. 443–458 (2020)
29. Ghavamnia, S., Palit, T., Mishra, S., Polychronakis, M.: Temporal System Call Specialization for Attack Surface Reduction. In: USENIX Security Symposium (SEC). pp. 1749–1766 (2020)
30. Ghavamnia, S., Palit, T., Polychronakis, M.: C2C: Fine-grained Configuration-driven System Call Filtering. In: ACM Conference on Computer and Communications Security (CCS). pp. 1243–1257 (2022)
31. Hawkins, B., Demsky, B.: ZenIDS: Introspective Intrusion Detection for PHP Applications. In: IEEE/ACM International Conference on Software Engineering (ICSE). pp. 232–243 (2017)

32. Huye, D., Shkuro, Y., Sambasivan, R.R.: Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In: USENIX Annual Technical Conference (ATC). pp. 419–432 (2023)
33. IBM: Microservices in the enterprise, 2021: Real benefits, worth the challenges. Tech. rep. (2021)
34. Istio Authors: Istio. https://istio.io (2024)
35. Jelesnianski, C., Ismail, M., Jang, Y., Williams, D., Min, C.: Protect the System Call, Protect (Most of) the World with BASTION. In: ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 528–541 (2023)
36. Jia, J., Zhu, Y., Williams, D., Arcangeli, A., Canella, C., Franke, H., Feldman-Fitzthum, T., Skarlatos, D., Gruss, D., Xu, T.: Programmable System Call Security with eBPF. arXiv preprint arXiv:2302.10366 (2023)
37. Jin, D., Gaidis, A.J., Kemerlis, V.P.: BeeBox: Hardening BPF against Transient Execution Attacks. In: USENIX Security Symposium (SEC) (2024)
38. Johannesmeyer, B., Slowinska, A., Bos, H., Giuffrida, C.: Practical data-only attack generation. In: USENIX Security Symposium (SEC). pp. 1401–1418 (2024)
39. Kallas, K., Zhang, H., Alur, R., Angel, S., Liu, V.: Executing Microservice Applications on Serverless, Correctly. Proceedings of the ACM on Programming Languages **7**, 367–395 (2023)
40. Kim, S., Kim, B.J., Lee, D.H.: Prof-gen: Practical Study on System Call Whitelist Generation for Container Attack Surface Reduction. In: IEEE International Conference on Cloud Computing (CLOUD). pp. 278–287 (2021)
41. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution. In: IEEE Symposium on Security and Privacy (S&P). pp. 93–101 (2020)
42. Kuenzer, S., Bădoiu, V.A., Lefeuvre, H., Santhanam, S., Jung, A., Gain, G., Soldani, C., Lupu, C., Teodorescu, c., Răducanu, C., Banu, C., Mathy, L., Deaconescu, R., Raiciu, C., Huici, F.: Unikraft: Fast, Specialized Unikernels the Easy Way. In: European Conference on Computer Systems (EuroSys). p. 376–394 (2021)
43. Li, X., Chen, Y., Lin, Z., Wang, X., Chen, J.H.: Automatic Policy Generation for Inter-Service Access Control of Microservices. In: USENIX Security Symposium (SEC). pp. 3971–3988 (2021)
44. Li, Y., Dolan-Gavitt, B., Weber, S., Cappos, J.: Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path. In: USENIX Annual Technical Conference (ATC). pp. 1–13 (2017)
45. Ling, Y., Rajiv, G., Gopinathan, K., Sergey, I.: Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis. In: USENIX Security Symposium (SEC). pp. 413–429 (2025)
46. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium (SEC). pp. 46–56 (2020)
47. Locust Authors: Locust.io. https://locust.io (2024)
48. Loukides, M., Swoyer, S.: Microservices Adoption in 2020. https://www.oreilly.com/radar/microservices-adoption-in-2020/
49. Luo, S., Xu, H., Lu, C., Ye, K., Xu, G., Zhang, L., Ding, Y., He, J., Xu, C.: Characterizing microservice dependency and performance: Alibaba trace analysis. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC). pp. 412–426 (2021)

50. Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., Huici, F.: My VM is Lighter (and Safer) than your Container. In: ACM Symposium on Operating Systems Principles (SOSP). pp. 218–233 (2017)
51. Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Gruss, D., Piessens, F.: Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: IEEE Symposium on Security and Privacy (S&P). pp. 1466–1482 (2020)
52. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In: ACM Conference on Programming Language Design and Implementation (PLDI). pp. 245–258 (2009)
53. National Institute of Standards and Technology: CVE-2013-2028. National Vulnerability Database, https://nvd.nist.gov/vuln/detail/CVE-2013-2028
54. Pailoor, S., Wang, X., Shacham, H., Dillig, I.: Automated Policy Synthesis for System Call Sandboxing. In: ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). pp. 1–26 (2020)
55. Provos, N.: Improving Host Security with System Call Policies. In: USENIX Security Symposium (SEC). pp. 257–272 (2003)
56. Provos, N., Friedl, M., Honeyman, P.: Preventing Privilege Escalation. In: USENIX Security Symposium (SEC). pp. 231–242 (2003)
57. Qian, C., Hu, H., Alharthi, M., Chung, P.H., Kim, T., Lee, W.: RAZOR: A Framework for Post-deployment Software Debloating. In: USENIX Security Symposium (SEC). pp. 1733–1750 (2019)
58. Rademacher, F., Sachweh, S., Zündorf, A.: Aspect-oriented Modeling of Technology Heterogeneity in Microservice Architecture. In: IEEE International Conference on Software Architecture (ICSA). pp. 21–30 (2019)
59. Rajagopalan, V.L., Kleftogiorgos, K., Göktas, E., Xu, J., Portokalidis, G.: SysPart: Automated Temporal System Call Filtering for Binaries. In: ACM Conference on Computer and Communications Security (CCS). p. 1979–1993 (2023)
60. Ren, Y., Liu, G., Nitu, V., Shao, W., Kennedy, R., Parmer, G., Wood, T., Tchana, A.: F2C: Enabling Fair and Fine-Grained Resource Sharing in Multi-Tenant IaaS Clouds. IEEE Transactions on Parallel and Distributed Systems **27**(9), 2589–2602 (2016)
61. Reselman, B.: 5 design principles for microservices (Sep 2023), https://developers.redhat.com/articles/2022/01/11/5-design-principles-microservices#conclusion
62. Saltzer, J.H., Schroeder, M.D.: The Protection of Information in Computer Systems. IEEE **63**(9), 1278–1308 (1975)
63. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: A Fast Address Sanity Checker. In: USENIX Annual Technical Conference (ATC). pp. 309–318 (2012)
64. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer – data race detection in practice. In: Workshop on Binary Instrumentation and Applications (WBIA). pp. 62–71 (2009)
65. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–10 (2010)
66. Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., Franz, M.: SoK: Sanitizing for Security. In: IEEE Symposium on Security and Privacy (S&P). pp. 1275–1295 (2019)
67. Sriraman, A., Wenisch, T.F.: µTune: Auto-Tuned Threading for OLDI Microservices. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 177–194 (2018)

68. StalkR's Blog: Universal go exploit using data races, no imports. https://blog.stalkr.net/2022/01/universal-go-exploit-using-data-races.html
69. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal War in Memory. In: IEEE Symposium on Security and Privacy (S&P). pp. 48–62 (2013)
70. Tang, A., Sethumadhavan, S., Stolfo, S.: CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: USENIX Security Symposium (SEC). pp. 1057–1074 (2017)
71. Tardieu, O., Grove, D., Bercea, G.T., Castro, P., Cwiklik, J., Epstein, E.: Reliable Actors with Retry Orchestration. ACM on Programming Languages **7**(PLDI), 1293–1316 (2023)
72. Team, M.: A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/
73. Team, T.C.: ThreadSanitizer – Clang 19.0.0git documentation. https://clang.llvm.org/docs/ThreadSanitizer.html
74. van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H., Giuffrida, C.: The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In: ACM Conference on Computer and Communications Security (CCS). pp. 1675–1689 (2017)
75. Vintila, E.Q., Zieris, P., Horsch, J.: Evaluating the Effectiveness of Memory Safety Sanitizers. In: IEEE Symposium on Security and Privacy (S&P). pp. 774–792 (2025)
76. Whitten, A., Tygar, J.D.: Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In: USENIX Security Symposium (SEC). pp. 169–184 (1999)
77. Zhang, H., Kallas, K., Pavlatos, S., Alur, R., Angel, S., Liu, V.: MuCache: A General Framework for Caching in Microservice Graphs. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 221–238 (2024)
78. Zhu, X., She, G., Xue, B., Zhang, Y., Zhang, Y., Zou, X.K., Duan, X., He, P., Krishnamurthy, A., Lentz, M., Danyang, Z., Mahajan, R.: Dissecting Overheads of Service Mesh Sidecars. In: ACM Symposium on Cloud Computing (SoCC). pp. 142–157 (2023)

A Appendix

Table 3: Syscalls allowed by filtering policies. R = Recording, S = Santa, SF = `sysfilter`, SP = `SysPart`, D = Docker. Omitted syscalls did not appear in any policy. CVE column lists one associated kernel vulnerability when known.

		Compose Post				Product Page				
syscall	**CVE**	**R**	**S**	**SF**	**SP**	**R**	**S**	**SF**	**SP**	**D**
accept	CVE-2017-8890	x	x	x	x			x	x	x
accept4						x	x	x		x
access		x	x	x	x	x	x	x	x	x
alarm								x	x	x
arch_prctl		x	x				x			x
bind	CVE-2016-10200	x	x	x	x	x	x	x	x	x
brk	CVE-2020-9391	x	x	x	x	x	x	x	x	x
capset										x
chdir								x	x	x
chmod	CVE-2016-7097							x	x	x
chown	CVE-2015-3339							x	x	x
clock_getres				x	x			x	x	x
clock_gettime	CVE-2011-3209			x	x			x	x	x
clock_nanosleep	CVE-2009-2767					x	x	x	x	x
clock_settime								x	x	
clone	CVE-2019-11815	x	x	x	x			x	x	x

Table 3 – continued from previous page

syscall	CVE	Compose R	Compose S	Compose SF	Compose SP	Post R	Post S	Post SF	Post SP	Product Page D
close		x	x	x	x	x	x	x	x	x
connect		x	x	x	x	x	x	x	x	x
dup	CVE-2016-3750			x				x	x	x
dup2								x	x	x
epoll_create	CVE-2011-1083									x
epoll_ctl	CVE-2013-7446						x			x
epoll_wait							x			x
eventfd2							x			x
execve	CVE-2018-14634					x	x	x	x	x
exit		x	x	x	x	x	x	x	x	x
exit_group				x	x			x	x	x
fadvise64								x	x	x
fcntl	CVE-2016-7118	x	x	x	x	x	x	x	x	x
fstat		x	x	x	x	x	x	x	x	x
ftruncate	CVE-2018-18281									x
futex	CVE-2020-14381	x	x	x		x	x	x	x	x
getcwd		x		x	x	x	x	x	x	x
getdents	CVE-2011-1593			x	x			x	x	x
getdents64						x	x	x	x	x
getegid						x	x	x	x	x
geteuid						x	x	x	x	x
getgid						x	x	x	x	x
getpeername				x	x					x
getpid			x	x	x	x	x	x	x	x
getppid						x	x	x	x	x
getrandom			x	x	x	x	x	x	x	x
getrlimit		x	x	x				x	x	x
getsockname	CVE-2021-38208	x	x	x	x	x	x	x	x	x
getsockopt	CVE-2021-20194			x	x	x	x	x	x	x
gettid			x	x	x	x	x	x	x	x
gettimeofday		x	x	x	x			x	x	x
getuid				x	x	x	x	x	x	x
ioctl	numerous drivers	x	x	x	x	x	x	x	x	x
kill			x	x	x			x	x	x
listen		x	x	x	x	x	x	x	x	x
lseek	CVE-2013-3301			x	x	x	x	x	x	x
lstat				x	x	x	x	x	x	x
madvise	CVE-2016-5195	x	x	x	x	x	x	x	x	x
mkdir								x	x	x
mmap	CVE-2018-7740	x	x	x	x	x	x	x	x	x
mprotect	CVE-2010-4169	x	x	x	x		x	x	x	x
mremap	CVE-2020-10757			x	x		x	x	x	x
munmap	CVE-2020-29369	x	x	x	x	x	x	x	x	x
nanosleep						x	x	x	x	x
newfstatat			x	x	x	x	x	x	x	x
open	CVE-2020-8428	x			x		x	x	x	x
openat	CVE-2020-10768			x	x	x	x	x	x	x
pause								x	x	x
pipe	CVE-2015-1805									x
poll		x	x	x	x	x	x	x	x	x
prctl	CVE-2020-10768									x
pread64										x
prlimit64		x		x		x	x	x	x	x
pselect6		x								x
pwrite64										x
pwritev										x
read		x	x	x	x	x	x	x	x	x
readlink	CVE-2011-4077			x	x	x	x	x	x	x
readv	CVE-2008-3535			x	x		x	x	x	x
recvfrom	CVE-2013-1979	x	x	x	x	x	x	x	x	x
recvmsg	CVE-2013-1979	x	x	x	x	x	x	x	x	x
rename	CVE-2016-6198							x	x	x
restart_syscall	CVE-2014-3180								x	x
rmdir								x	x	x
rt_sigaction		x	x	x	x	x	x	x	x	x
rt_sigprocmask		x	x	x	x	x	x	x	x	x
rt_sigreturn	CVE-2017-15537			x	x			x	x	x
rt_sigsuspend					x					x
sched_get_priority_max		x	x	x	x	x	x	x	x	x
sched_get_priority_min		x	x	x	x	x	x	x	x	x
sched_getaffinity				x	x	x	x	x	x	x
sched_getparam								x	x	x
sched_getscheduler								x	x	x
sched_setaffinity	CVE-2021-26708							x	x	x
sched_setscheduler								x	x	x
sched_yield							x	x	x	x
select			x		x					x
sendfile								x	x	x
sendmmsg	CVE-2011-4594	x	x	x		x	x	x	x	x
sendmsg	CVE-2017-17712					x	x	x	x	x
sendto	CVE-2017-17712	x	x	x	x	x	x	x	x	x
set_robust_list		x	x	x	x	x	x	x	x	x
set_tid_address		x	x	x		x	x	x	x	x
setgid	CVE-2021-32760			x		x	x	x	x	x
setgroups	CVE-2018-7169			x		x	x	x	x	x
setitimer					x			x	x	x
setpriority								x	x	x

Table 3 – continued from previous page

syscall	CVE	Compose R	Compose S	Compose SF	Compose SP	Post Product R	Post Product S	Post Product SF	Post Product SP	Page D
setregid						x	x	x	x	x
setresgid		x	x	x		x	x	x	x	x
setresuid	CVE-2019-18684	x	x	x		x	x	x	x	x
setreuid	CVE-2011-3145			x		x	x	x	x	x
setrlimit		x	x					x	x	x
setsid	CVE-2005-0178			x	x					x
setsockopt	CVE-2016-4998	x	x	x	x	x	x	x	x	x
setuid	CVE-2013-6825			x		x	x	x	x	x
shmat	CVE-2017-5669				x					x
shmdt					x					x
shmget	CVE-2017-5669				x					x
shutdown		x	x	x	x	x	x	x	x	x
sigaltstack	CVE-2009-2847						x	x	x	x
socket	CVE-2017-9074	x	x	x	x	x	x	x	x	x
socketpair	CVE-2010-4249	x	x			x	x	x	x	x
stat		x	x	x	x	x	x	x	x	x
statfs										x
sysinfo				x	x	x	x	x	x	x
tgkill	CVE-2013-2141			x	x	x		x	x	x
time						x	x	x	x	x
times							x			x
umask	CVE-2020-35513					x	x	x	x	x
uname	CVE-2012-0957	x	x	x	x	x	x	x	x	x
unlink	CVE-2016-6197							x	x	x
utimes								x	x	x
vfork	CVE-2005-3106			x				x	x	x
wait4		x	x			x	x	x	x	x
write		x	x	x	x	x	x	x	x	x
writev	CVE-2016-9755			x	x	x	x	x	x	x

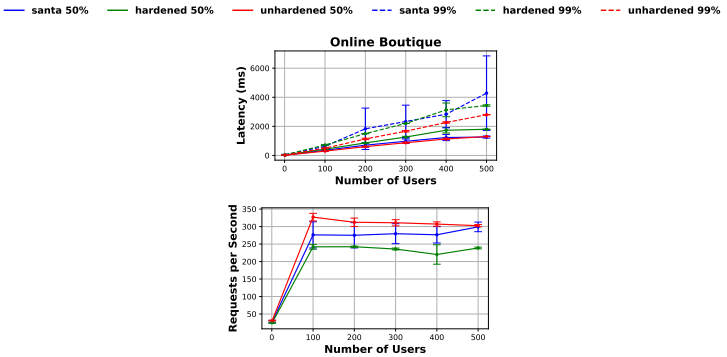

Fig. 4: Comparison of latency and requests per second of Online Boutique execution with Santa, hardened, and unhardened systems.