Secure Namespaced Kernel Audit for Containers

Soo Yee Lim
University of British Columbia
Vancouver, British Columbia, Canada
sooyee@cs.ubc.ca

Bogdan Stelea
University of Bristol
Bristol, United Kingdom
bs17580@bristol.ac.uk

Xueyuan Han
Harvard University
Cambridge, Massachusetts, USA
hanx@g.harvard.edu

Thomas Pasquier
University of British Columbia
Vancouver, British Columbia, Canada
tfjmp@cs.ubc.ca

ABSTRACT

Despite the wide usage of container-based cloud computing, container auditing for security analysis relies mostly on built-in host audit systems, which often lack the ability to capture high-fidelity container logs. State-of-the-art reference-monitor-based audit techniques greatly improve the quality of audit logs, but their system-wide architecture is too costly to be adapted for individual containers. Moreover, these techniques typically require extensive kernel modifications, making it difficult to deploy in practical settings.

In this paper, we present saBPF (**s**ecure **a**udit **BPF**), an extension of the eBPF framework capable of deploying secure system-level audit mechanisms at the container granularity. We demonstrate the practicality of saBPF in Kubernetes by designing an audit framework, an intrusion detection system, and a lightweight access control mechanism. We evaluate saBPF and show that it is comparable in performance and security guarantees to audit systems from the literature that are implemented directly in the kernel.

CCS CONCEPTS

• **Security and privacy** → *Operating systems security*; Intrusion detection systems; • **Networks** → *Cloud computing*.

KEYWORDS

eBPF, auditing, container, provenance

SoCC '21, November 1–4, 2021, Seattle, WA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486976

ACM Reference Format:
Soo Yee Lim, Bogdan Stelea, Xueyuan Han, and Thomas Pasquier. 2021. Secure Namespaced Kernel Audit for Containers. In *Proceedings of ACM Symposium on Cloud Computing (SoCC '21)*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472883.3486976

1 INTRODUCTION

In recent years, container-based cloud computing has gained much traction. As a lightweight alternative to VM-based computing infrastructure, it provides an attractive multi-tenant environment that supports the development of microservice architecture, where a monolithic application is organized into a number of loosely-decoupled services for modularity, scalability, and fault-tolerance [59].

Container security becomes a major concern as the popularity of container-based cloud continues to grow rapidly [64]. For example, by sharing individual microservices across applications, the container ecosystem, as promoted by widely-used container management and orchestration platforms, such as Docker and Kubernetes, inadvertently spreads vulnerabilities that can widen an application's attack surface [63]. Vulnerabilities in individual containers can facilitate the construction of a *cyber kill-chain*, in which the attackers perform various attacks in steps on different microservices to achieve their ultimate goal [37]. Information leakage between a host and a container and between two co-resident containers has also been demonstrated to be possible [25].

Like in traditional security analysis, system-layer audit logs are often considered to be an important information source for addressing many container security concerns [29, 31, 44]. For example, to identify a misbehaving container from a cluster of replicated microservices, kernel audit logs have been used to define process activity patterns and describe unusual activity that does not fit into any observed pattern [32]. Container-focused security systems typically use existing host audit tools, such as the Linux Audit Framework, to log system events, but research has shown that these audit systems are insufficient to capture complete system activity necessary for security analysis [26]. Alternative *reference-monitor-based* approaches [51, 54] provide a more complete picture by leveraging in-kernel monitoring hooks, but they are not designed with container-based computing architecture in mind. Specifically, reference-monitor-based approaches require extensive host kernel modification and permit only host-wide policy specification. The former requirement is often forbidden by cloud infrastructure providers, and the latter makes it difficult to satisfy the audit needs of individual containers sharing the same host.

We present SABPF, a lightweight, secure kernel audit system for containers. SABPF enables each container to customize its auditing mechanism and policy, even if containers specifying different policies and mechanisms are co-located on the same host. Audit data captured by SABPF is guaranteed to have *high fidelity*, meaning that the data *faithfully* records *complete* container-triggered system activity [54], free of concurrency vulnerabilities [66] and missing records [26] that are commonly present in existing audit tools. As such, SABPF builds a solid foundation for future forensic applications in a containerized cloud environment. For example, to deploy a cyber kill-chain detection system for a network of Kubernetes containers (or *pods*), we can follow the *Sidecar* design pattern, in which SABPF is configured for each pod based on the characteristics of the microservice it provides. A specialized sidecar container is attached to capture and analyze audit logs. Sidecar containers from different pods can ship any suspicious events to a remote system where alert correlation is performed to detect the presence of a kill-chain [45]. We discuss this use case in more detail in § 4.

SABPF is implemented as an extension of eBPF. Specifically, we make the following contributions:

- We expand Linux's eBPF framework to support the attachment of eBPF programs at the intersection of the reference monitor and namespaces, which allows fully-configurable, high-fidelity, system-level auditing for individual containers (see § 3);
- We develop functional proof-of-concept applications using SABPF to demonstrate its practicality (see § 4);
- We conduct thorough performance analysis to understand the cost and benefits of using SABPF for secure audit; we show that our approach outperforms existing audit solutions of similar caliber (see § 5 and § 6);
- We open source SABPF to facilitate the development of security applications for containers in the cloud (see Appendix A).

2 BACKGROUND

SABPF is built upon the *extended Berkeley Packet Filter* (eBPF) framework, a Linux subsystem that allows user-defined programs to safely run inside the kernel [8]. To provide high-fidelity system audit data, SABPF implements eBPF programs

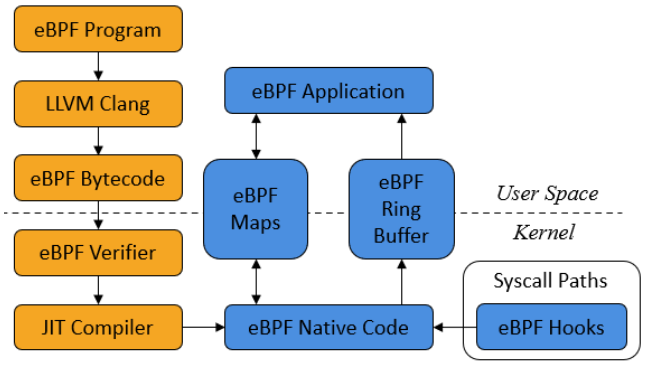

Figure 1: eBPF workflow.

that instrument in-kernel hooks defined by the *Linux Security Modules* (LSM) framework, which is the reference monitor implementation for Linux [46]. SABPF further leverages *namespaces* to ensure that auditing can be customized for individual containers. We provide some background knowledge for each component.

2.1 Extended Berkeley Packet Filter

eBPF is a Linux built-in framework that allows customized extensions to the kernel without modifying the kernel's core trusted codebase. We illustrate how eBPF works in Fig. 1. Developers can write an eBPF program in C and compile the program into eBPF bytecode using Clang. The kernel uses a verifier to statically analyze the bytecode, minimizing security and stability risks of running untrusted kernel extensions [28]. After verification, a just-in-time (JIT) compiler dynamically translates the bytecode into efficient native machine code. The translated eBPF program is attached to designated kernel locations (e.g., LSM hooks in our case) and executed at runtime. eBPF programs can share data with user-space applications using special data structures such as eBPF maps and ring buffers [3].

eBPF is frequently used as the underlying framework for network security and performance monitoring. In a container environment, for example, eBPF enables Cilium [5], a popular network monitoring tool for platforms such as Kubernetes, to secure application-level protocols with fine-grained firewall policies.

2.2 Linux Security Modules

LSM consists of a set of in-kernel *hooks* that are strategically placed where kernel objects (such as processes, files, and sockets) are being accessed, created, or destroyed. LSM hooks can be instrumented to enable diverse security functionality, with the canonical usage being the implementation

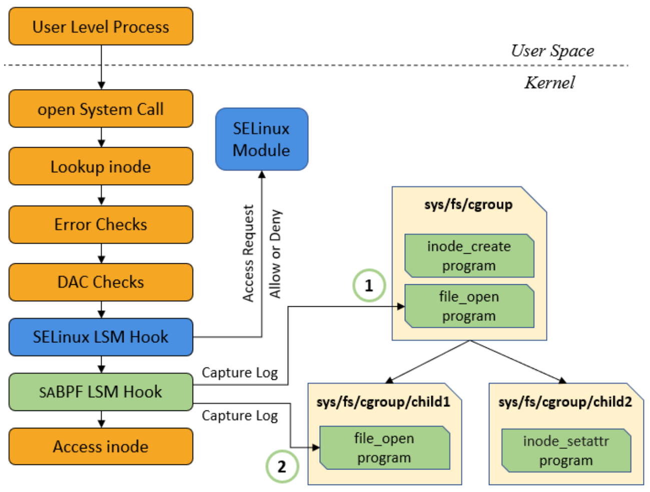

Figure 2: LSM hook architecture [46]. The green blocks are specific to SABPF, which is described in § 3. ① shows programs attached to the root cgroup; ② shows two programs attached to the child1 and child2 cgroup respectively.

Name	Description
cgroup	Allocate system resource (e.g., CPU, memory, and networking)
ipc	Isolate inter-process communications
network	Virtualize the network stack
mount	Control mount points
process	Provide independent process IDs
user	Provide independent user IDs and group IDs, and give privileges (or capabilities) associated with those IDs within other namespaces
UTS	Change host and domain names
Time	See different system times

Table 1: Summary of Linux namespaces.

of mandatory access control scheme [56]. As a reference monitor, LSM has also been adapted to perform secure kernel logging, which provides stronger *completeness* and *faithfulness* guarantees than traditional audit systems [17, 51, 54]. For example, prior research has verified that LSM hooks capture all meaningful interactions between kernel objects [22, 36] and that information flow within the kernel can be observed by at least one LSM hook [27], which is necessary to achieve completeness.

The LSM framework does not use system call interposition as older systems did. Syscall interposition is susceptible to concurrency vulnerabilities, which in turn lead to time-of-check-to-time-of-use (TOCTTOU) attacks that result in discrepancies between the events as seen by the security mechanism and the system call logic [66, 67]. This is why solutions such as kprobe-BPF, while useful for performance analysis, are not appropriate to build security tools. Instead, LSM’s reference-monitor design ensures that the relevant kernel states and objects are immutable when a hook is triggered, which is necessary to achieve faithfulness. LSM-BPF [57] is a recent extension to the eBPF framework that provides a more secure mechanism to implement security functionalities on LSM hooks.

2.3 Namespaces in Linux

A namespace in the Linux kernel is an abstract environment in which processes within the namespace appear to own an independent instance of system resources. Changes to those resources are not visible to processes outside the namespace. We summarize available namespaces in Linux in Table 1.

One prominent use of namespaces is to create containers. For example, an application in a Docker container runs within its own set of namespaces. Kubernetes “pods” contain one or more containers so that they share namespaces (and therefore system resources). Kubernetes makes it appear to applications within a pod that they own a machine of their own (Fig. 3).

SABPF modifies the kernel to enable per-container auditing, selectively invoking eBPF programs on LSM hooks based on cgroup membership (§ 3.1). cgroup isolates processes’ resource usage in a hierarchical fashion, with a child group having additional restrictions to those of its parent. Since cgroup v2 [35], this hierarchy is system-wide, and all processes initially belong to the root cgroup. In a Kubernetes pod, for example, containers can be organized in a hierarchical structure and assigned various cgroup namespaces to set up further restrictions. Certain types of eBPF programs, such as the ones that are socket-related, can already be attached to cgroups (e.g., BPF_PROG_TYPE_CGROUP_SKB). This allows, for example, a packet filtering program to apply network filters to sockets of all processes within a particular container. SABPF makes it possible to attach eBPF programs at the intersection of cgroups and LSM hooks (§ 3.1) for audit purpose and beyond (§ 4).

3 SABPF: EXTENDING THE EBPF FRAMEWORK

SABPF extends the eBPF framework to support secure kernel auditing in a containerized environment. Our design is *minimally invasive*, reusing existing components in the framework as much as possible and extending only what we deemed to be necessary. This is a conscious decision made to achieve two objectives: 1) by adhering closely to the

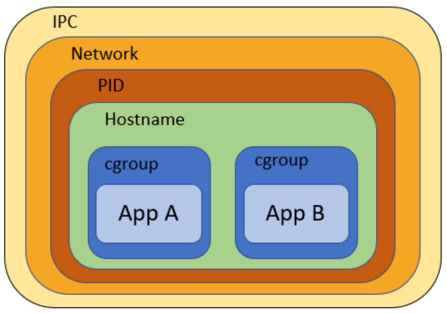

Figure 3: Namespaces in Kubernetes.

design philosophy of the eBPF framework, we ensure that saBPF can be readily integrated into the mainline kernel; 2) since saBPF is built upon the eBPF framework and adheres to its design philosophy, users already familiar with eBPF can quickly develop new applications using saBPF, while new users have access to eBPF's documentations and forums, which makes saBPF easy to learn and use. In addition, any existing eBPF program can run in conjunction with our audit solution on the saBPF-enhanced platform for individual containers.

3.1 Namespacing LSM-BPF

saBPF extends the use of `cgroup` (which in eBPF is used mostly for network filtering) to LSM hooks. This extension allows saBPF to precisely control audit granularity. Recall that `cgroup`s are arranged in a global hierarchy since `cgroup` v2, and all processes belong to the root `cgroup` by default. A Kubernetes pod, for example, defines a *pod-level* `cgroup`, which is the ancestor of *container-level* `cgroup`s within which individual containers in the pod reside (Fig. 3). By attaching eBPF programs to container-level `cgroup`s, saBPF can perform container-level auditing; at the same time, saBPF can monitor activity inside the entire pod by attaching audit programs to the pod's root `cgroup`. While the design of Kubernetes makes it natural to follow this two-level audit scheme using `cgroup`, saBPF can support arbitrarily complex `cgroup` hierarchy for customizable use.

Listing 1 illustrates how an saBPF program is defined by developers. It is a program that simply prints "Hello World!" when the `file_open` LSM hook is triggered. The statement in line 1 specifies where the program should be attached, and the `ctx` variable in line 2 contains the parameters passed from the `file_open` hook. The user attaches (and detaches) this program to (and from) a `cgroup` through the `bpf()` system call. Multiple eBPF programs can be attached to the same `cgroup`-hook pair by setting the `BPF_F_ALLOW_MULTI` flag; they will be executed in FIFO order.

```
1  SEC("cgroup_lsm/file_open")
2  int cgroup_file_open(struct bpf_cgroup_lsm_ctx *
       ctx)
3  {
4    bpf_trace_printk("Hello World!\n");
5    return 0;
6  }
```

Listing 1: A "Hello World!" saBPF program that can be triggered on the `file_open` hook.

In the kernel, when an LSM hook is triggered, saBPF invokes appropriate eBPF programs through a customized security module. This module performs three main actions for every hook. First, it prepares the parameters, which are passed from the hooks to the eBPF programs. It then retrieves the `cgroup` associated with the current task. Finally, it traverses the `cgroup` hierarchy (§ 2.3) backwards from the current `cgroup` to the root `cgroup` and executes all programs associated with the LSM hook.

Fig. 2 illustrates this process using the `open` system call as an example. The root `cgroup` has two child `cgroup`s, `child1` and `child2`. While the root `cgroup` has programs attached to both `inode_create` and `file_open` LSM hooks, `child1` has only one program attached to `file_open` and `child2` has one program attached to a different LSM hook, `inode_setattr`. As a result, *any* process that triggers the `file_open` hook leads to the invocation of the programs attached to that hook in the root `cgroup`. However, the programs attached to the same hook in `child1` are only called if a process belonging to `child1` (or one of its decedents) triggers the hook. Note that this process still causes programs in the root `cgroup` to be called.

Early in the design phase, we considered creating a dedicated namespace for system auditing. While this allows a clear separation of namespaces' responsibilities, given that `cgroup`s are designed to control access to system resources, we eventually abandoned this design for two reasons. First, significant re-engineering of existing container solutions would be required to make use of this new namespace. Second, existing namespaced eBPF already uses `cgroup`. We believe that introducing a new namespace goes against the current design philosophy of eBPF. However, we emphasize that based on our experience, it is relatively straightforward to implement a new namespace should such a need arise in the future.

3.2 Local Storages

System auditing often requires associating data with kernel objects [54]. In early prototypes, we considered using

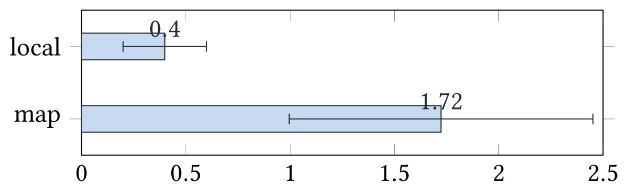

Figure 4: Look-up time for the `cred` local storage and the eBPF map in μs. Using local storage gives a *4x* speedup.

eBPF maps, which are key-value stores shared among multiple eBPF programs across execution instances, but we abandoned the idea due to poor maintainability. Specifically, when using eBPF maps, developers must create an entry for each new kernel object to store data associated with the object. The key to the entry must be unique during the lifecycle of the object. Ensuring uniqueness for all kernel objects is important, but prone to error. For example, it is insufficient to use just an inode number as the key for an `inode` object; rather, a *combination* of the inode number and the file system's unique identifier is needed because inode numbers are guaranteed to be unique *per file system* only. Moreover, developers must also take special care to remove map entries when objects reach the end of their lifecycle. This problem is exacerbated by the fact that eBPF maps are created with capacity limits.

saBPF uses a completely different approach to storing such data, extending eBPF's *local storages*, which are data structures that are directly associated with kernel objects. Local storages provide an interface similar to eBPF maps, but they use the object reference as the key and store the value locally with the kernel object. At the end of an object's lifecycle when the object no longer has any reference, eBPF transparently removes the local storage associated with the object. This takes the responsibility of removing unused entries away from developers, making it less error-prone. Furthermore, local storages incur less performance overhead compared to eBPF maps, as shown in Fig. 4. At the time of writing, eBPF provides local storages for only `cgroup`, `socket`, `inode` and `task`. We implemented additional local storages for `file`, `cred`, `ipc`, `superblock`, and `msg_msg` to *fully* support LSM-based auditing. We give a practical illustration in § 4.1.

3.3 Extension of eBPF interface

To access kernel data, eBPF programs rely on *eBPF helpers*, which are an allowlist of kernel functions permitted by the eBPF verifier to interact with the kernel. saBPF defines a number of extra eBPF helpers, as shown in Table 2, to facilitate system auditing.

Name	Description
`bpf_inode_from_sock`	Retrieve the inode associated with a socket
`bpf_file_from_fown`	Retrieve the file associated with a `fown_struct`
`bpf_dentry_get`	Retrieve the `dentry` associated with an inode
`bpf_dentry_put`	Release a `dentry` after use
`bpf_[cred/msg/ipc/file]_storage_get`	Get a `bpf_local_storage` from a `cred/msg/ipc/file`
`bpf_[cred/msg/ipc/file]_storage_delete`	Delete a `bpf_local_storage` from a `cred/msg/ipc/file`

Table 2: Summary of new eBPF helpers provided by saBPF.

A subset of functions return the inode associated to an object of a certain type (e.g., `bpf_inode_from_sock` returns the inode of a socket object). This can be useful to understand the interplay of system calls acting at different levels of kernel abstraction.

`bpf_dentry_get` returns the directory entry of an inode, which helps saBPF programs to retrieve the path associated with the inode. A directory entry is protected by a reference counter when a program manipulates it. The reference counter is increased when `bpf_dentry_get` is called and *must* be decreased by calling `bpf_dentry_put` once the entry is no longer used. To ensure correctness, we also modified the eBPF verifier to verify that every `bpf_dentry_get` has a corresponding `bpf_dentry_put` being called on the same code path.

The remaining helpers are used to manipulate local storages (§ 3.2). We extended eBPF map helpers so that userspace programs can interact with those storages. In eBPF maps, userspace programs can access a set of helpers via system calls to e.g., update or lookup map entries. Our extension provides similar support for local storages: programs can manipulate data in the local storage of a particular kernel object using appropriate userspace identifiers. For example, assuming appropriate privileges, we can lookup, update, and delete data in a `cred`[1] object's local storage via its `PID`.

4 USE CASES

A framework like saBPF is only useful if it is both *practical* and *performant*. We discuss three meaningful use cases that we have implemented to demonstrate the types of application that saBPF can easily support, showcasing its practicality. We evaluate saBPF's performance in § 6.

[1]`cred` is the credential information associated with a process.

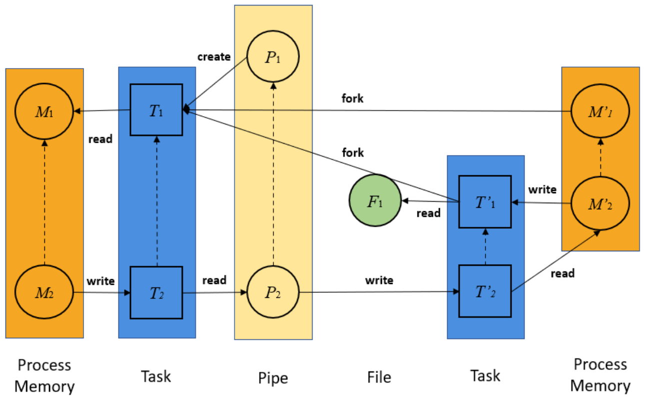

Figure 5: A simplified whole-system provenance subgraph.

4.1 Whole-system Provenance Capture

We describe our implementation of ProvBPF, a provenance capture mechanism that we developed atop saBPF. Provenance has gained much traction in the security community, notably with applications designed to understand intrusions in a computer system [33, 34, 44], prevent data exfiltration [17], and detect attacks [29–31, 42, 45, 65]. ProvBPF captures provenance at the thread granularity, recording information such as security context, namespace, and performance metrics.

A Brief Provenance Introduction. Computing systems are too often opaque: they accept inputs and generate outputs, but the visibility of their inner workings is at best partial, which poses many issues in fields ranging from algorithmic transparency to the detection of cybersecurity threats. Unfortunately, traditional tracing mechanisms are inadequate in addressing these issues. Instead, *whole-system provenance* [54], which describes system execution by representing information flows within and across systems as a directed acyclic graph, shows promise. Provenance records subsume information contained in a traditional trace, while causality relationships between events can be inferred through graph analysis.

Fig. 5 shows a simple provenance graph. In this graph, two tasks (T and T') are associated with their respective memory (M and M'). The subscripts (e.g., T_1 and T_2) represent different *versions* of the same kernel object to guarantee graph acyclicity [48]. T creates a pipe P and forks a new process (corresponding to T' and M'). T' reads information from a file F and writes information to P. T then reads from this pipe. A versioned node is created every time an object receives external information (e.g., when a task reads from a file). This is a small subgraph representing a very simple scenario. In practice, for example, a graph representing the compilation of the Linux kernel would contain approximately a few million graph elements [47].

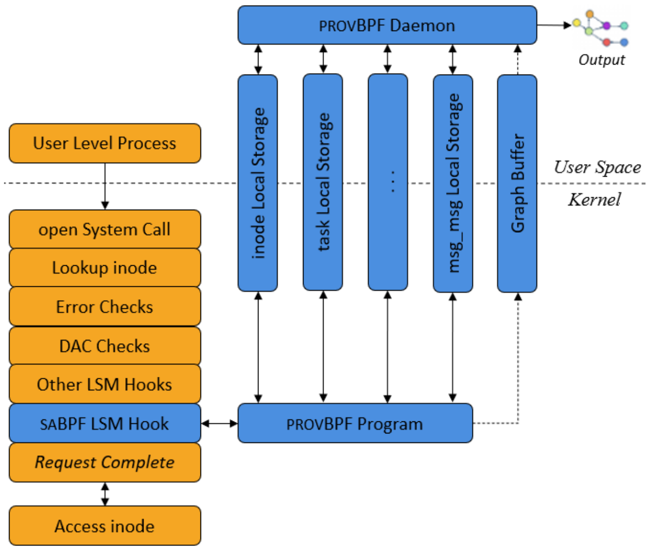

Figure 6: ProvBPF overview.

The rest of our discussion focuses primarily on the novel aspects of ProvBPF and the design choices we made as the result of using saBPF to capture OS-level provenance (instead of modifying the kernel). We compare ProvBPF's performance to that of a state-of-the-art provenance capture system, CamFlow [51], in § 6.

Overview. Fig. 6 illustrates the architecture of ProvBPF using the open system call as an example. In ProvBPF, eBPF programs are executed on LSM hook invocations. ProvBPF's eBPF programs generate provenance graph elements in binary and write them to an eBPF ring buffer [3]. A user-space daemon serializes those graph elements and outputs them to disk (or to remote endpoints like Apache Kafka [1]) in a machine readable format such as W3C PROV-DM [18].

We must associate *states* with kernel objects to guarantee graph acyclicity and to implement graph compression algorithms. For example, to guarantee acyclicity, we associate with each object a version counter which is updated when external information flows into an object and modifies its state. After the update, a new vertex is added to the graph and connected to the previous version of the object through a `version` edge, as illustrated in Fig. 5 as dashed lines.

We also associate an "opaque" flag to the state of certain kernel objects; opaque objects are not audited. This is particularly useful for ProvBPF's daemon-related objects, because capturing their provenance would result in an infinite feedback loop. CamFlow uses `security blobs` from the LSM framework associated with each kernel object to maintain its associated states. In ProvBPF, we leverages the *local storage* mechanism (§ 3) for this purpose. Local storage can be accessed by the ProvBPF daemon from userspace to set a policy for each individual object (e.g., to set opacity).

Graph Reduction. eBPF-based provenance capture offers exceptional flexibility in designing customized capture policies that fulfill different objectives. Customization typically involves *filtering*, i.e., selecting kernel objects and system events that are relevant to a specific analysis. For example, Bates et al. [15] only record events related to objects associated with specific SELinux policies. ProvBPF allows for the filter logic to be built-in during compilation, thus reducing run-time overhead.

ProvBPF implements additional graph reduction techniques other than filtering. It automatically merges consecutive events of the same type between two entities into a single event and avoids object versioning as much as possible. Event merging reduces the number of edges between two nodes without changing the semantics of the interactions they represent. For example, when a process reads a file piece-by-piece through a number of successive `read` system calls, saBPF would create only one directed edge between the process and the file to capture these read events, which is sufficient to describe the information flow from the file to the process due to read. On the other hand, avoiding object versioning reduces the number of nodes, and ProvBPF does so only when the semantics of an object have not changed. These graph reduction techniques are completely agnostic to specific downstream provenance analysis and can be easily configured at compilation time according to the needs of a particular application. More importantly, unlike previous work [61] that performs graph compression as a costly post-processing step (i.e., after recording the original graph), ProvBPF employs those techniques during capture *before* new edges are added to the graph.

Verifying Capture Correctness. It is challenging to verify the correctness of a provenance capture mechanism [19]. At a minimum, we must show that a provenance graph describing system activity of a system call makes "intuitive" sense for a human analyst inspecting the graph. We use both static and dynamic analysis to verify that provenance graphs generated by ProvBPF are reasonably correct.

Our static analysis generates a graph motif for each system call, which enables us to reason about the semantics of the graph based on our understanding of the system call. We follow the same strategy as described by Pasquier et al. [52]. To generate a system call graph motif, we first analyze the kernel codebase to construct a call graph of a system call and extract a subgraph, within the call graph, that contains only LSM hooks [27]. We then analyze ProvBPF's codebase to generate a graph motif for each LSM hook and augment the subgraph from the previous step by replacing each LSM hook in the subgraph with the corresponding graph motif. The resulting subgraph – now containing only graph motifs of LSM hooks – is the graph motif of the system call that summarizes what the provenance graph would look like when the system call is executed.

In addition, we build test programs and follow the same steps above to create program-level graph motifs. For each program binary, we build a call graph which we then transform into a syscall-only graph. We replace the syscalls in the graph with the motifs we previously built. We run each test program and verify by inspection that our (statically-produced) motif matches the (dynamically-produced) provenance graph generated by ProvBPF. We perform the same steps in CamFlow [51] to verify that the graphs generated by the two systems are equivalent.

4.2 An Intrusion Detection System for Kubernetes

saBPF-based audit systems such as ProvBPF can be used as an underlying framework for various security applications in the cloud. We demonstrate this feasibility through a concrete use case of deploying Unicorn [29], a state-of-the-art host-based intrusion detection system (IDS), in a Kubernetes pod using ProvBPF as an upstream information provider. Unicorn is an anomaly-based IDS that learns system behavior from the provenance graph generated by benign system activity. Once a model is learned from the graph, detection is formulated as a graph comparison problem: if a running system's provenance graph deviates significantly from the model, Unicorn considers the system to be under attack. In the remainder of this section, we focus our discussion on how ProvBPF facilitates deployment of an IDS in a containerized environment in a novel and elegant manner; in-depth evaluation of the performance of such an IDS is out of scope and left for future work.

Design & Implementation. ProvBPF makes it easy to run a provenance-based IDS at the pod level in Kubernetes, which is challenging when provenance data is provided by a reference-monitor-based audit system such as CamFlow [51]. For systems like CamFlow, provenance is always captured *system-wide*; as a result, audit logs must be filtered to provide as input to the IDS provenance data relevant to a pod only, and filtering must be done on an individual pod basis. Unfortunately, this extra filtering step inevitably adds delay and complexity to the entire detection pipeline, thus reducing *runtime* detection efficacy.

Instead, we use ProvBPF and Kubernetes' sidecar design pattern to attach the IDS to Kubernetes applications. A sidecar container is a container that runs alongside a main container (i.e., the one that provides core functionality) in a pod. In our design, for each pod, we include a sidecar container that runs both ProvBPF and Unicorn. ProvBPF audits the entire pod and generates a pod-level provenance graph; the graph is then used as input to Unicorn. We note that other detection systems such as StreamSpot [42] and log2vec [41] could be used in a similar fashion. In a microservice environment, any misbehavior detected within a single pod can be sent to a dedicated central service that performs alert correlation [45] to detect, for example, early stages of a cyber kill-chain.

Discussion. This deployment strategy, made possible by ProvBPF, have a number of advantages. First, we do not need to modify applications to deploy our IDS thanks to the sidecar pattern. Second, we can easily deploy an IDS model specific to an application running in a pod, without taking into consideration extraneous activity of the rest of the system. Third, ProvBPF produces provenance graph elements that can be analyzed directly, without introducing filtering delays in the detection pipeline. Fourth, deploying ProvBPF imposes no cost on other pods running on the same machine, since saBPF programs are only triggered within the context of a single pod. This is in contrast to a classic system-wide approach (e.g., Linux Auditd or CamFlow), which would negatively affect performance on the entire machine.

4.3 Lightweight Ad-hoc Access Control

While saBPF was designed primarily to provide secure auditing, it can also be used to implement simple access control policy within the scope of a container.[2] We implemented a proof-of-concept to demonstrate saBPF's ability to enforce access control policy. Like in § 4.2, we consider a Kubernetes environment and use the sidecar pattern to deploy access control policy at the *pod* granularity.

Design & Implementation. Using saBPF, we can easily achieve *separation of concerns* in Kubernetes, such that each pod has its own set of security mechanisms and policies. We deploy a sidecar alongside unmodified applications to constrain their behavior. The sidecar runs a set of saBPF programs implementing the desired policy and attach them to the root `cgroup` of the pod. We associate security contexts to kernel objects through local storage and define an eBPF map to store constraints applicable to those contexts. When an LSM hook is triggered, information is retrieved from the map to determine whether or not an action is permitted. Policy violation can be sent to userspace via an eBPF ring buffer, which can then be logged or reported to the user about the unexpected application behavior.

We take advantage of the eBPF framework to optimize the sidecar at the time of its compilation based on the policy to be enforced. For example, if the policy has no network access rules, we do not build any rules enforcement program regarding network access. In general, our access control mechanism generates a minimum set of programs needed to enforce a given policy, thus reducing complexity and improving performance.

Policy Example. Taking inspiration from the Open Policy Agent [12] and AppArmor [2], we create a simple policy language. A policy is expressed in JSON and parsed to generate a customized sidecar application that can be attached to a pod.

```
{
  "target": "/usr/bin/foo",
  "network": {"default": "deny", "allow_egress": [404,80]},
  "file": {"default": "r", "rw": ["/tmp/*"], "mr": ["/lib/ld-*.so*",
   "/lib/lib*.so*"]}
}
```

Listing 2: A simple policy example.

Listing 2 shows an exemplar policy written in this language. A `/usr/bin/foo` process is by default denied access to the network unless it is an outgoing connection through the `http` and `https` ports. Similarly, the process is denied `write` or `execute` by default. However, it has read and write access to the `/tmp` directory and is able to map system libraries. This policy is inherited by any child process `forked` from the `/usr/bin/foo` process.

4.4 Discussion

We conclude this section by summarizing the advantages of using saBPF, as repeatedly demonstrated by the three use cases described above.

Performance. saBPF is the first reference-monitor-based audit system that allows audit rule configuration at compilation time, drastically minimizing run-time audit complexity and improving overall performance. In stark contrast, other audit mechanisms such as CamFlow must evaluate complex audit rules at runtime to satisfy specific needs of different downstream applications. For example, security tools such as SIGL [31] typically analyze only a small subset of host activity logs that an audit system like CamFlow provides. To monitor an application in a Kubernetes pod, Unicorn requires provenance data generated only by activity in the pod. In both cases, filtering is inevitable but it can sometimes become a performance bottleneck that is difficult to overcome. To make matters worse, as we have discussed in § 5.1, run-time evaluation can have adverse and cumulative performance impact, making existing reference monitors undesirable to be even considered in practical settings. Similarly, a given application may only enforce access control constraints on

[2]Policy conflict resolution across `cgroup` hierarchy is out of scope of this paper. We refer interested readers to § 8.

a subset of events. Through compilation-time policy evaluation, saBPF can minimize run-time cost by running only needed eBPF programs.

In practice, this means that given an equivalent policy, a solution built using saBPF significantly outperforms current solutions developed through the built-in LSM mechanism.

Maintainability and Adoption. Maintaining out of tree LSMs requires significant effort and time investment. As LSMs are built in the kernel, rigorous testing is essential to avoid crashes or introducing unintended security vulnerabilities. This makes exploring new mechanisms difficult. For similar reasons, it is rare for third parties to further develop on a custom kernel given the high risk of instability and vulnerability. We further discuss maintainability concerns in § 7.

Decentralized Deployment. Standard LSM-based solutions are generally deployed system-wide and centralized, and must be managed by the host. By contrast, each containerized environment (assuming proper privileges) can deploy its own LSM mechanisms using saBPF without affecting the rest of the system. Each guest environment can run not only different policies, but also a completely different mechanism. Moreover, saBPF programs are only triggered within the cgroup they are attached to, thus limiting data leakage across containers (see § 7 for further discussion on security).

System Call	Security Hooks	Min Hook Calls
open	file_open+ inode_create inode_permission* inode_post_setxattr inode_setattr	1 + 1 × path depth
read	file_permission+	1
write	file_permission+	1
execve	bprm_check+ bprm_set_creds+ file_open+ inode_permission* file_permission+	4 + 1 × path depth

Table 3: Summary of LSM hooks called on successful system calls. Some hooks are only triggered in a particular system state or with specific syscall parameters (e.g., when creating a *new* file on open). + indicates that hooks are always called, and * means hooks are called on every directory in a path.

5 UNDERSTANDING POLICY OVERHEAD

The run-time performance overhead of any always-on audit system is critical to its successful adoption. While the overall performance is a function of a specific audit policy, which varies across different needs and use cases, the run-time cost of the underlying infrastructure can be reasonably analyzed, which we present in this section. Our analysis focuses on two main components of saBPF, LSM and eBPF; the cost of running both together has not been widely studied, especially in the context of audit. We also compare our approach with state-of-the-art reference-monitor-based auditing that requires kernel modification.

5.1 LSM overhead

It is difficult to precisely measure LSM overhead [68]. In general, there exist two sources of overhead when performing audit (or other policy enforcement) through LSM: *hooking* and *execution*. Hooking refers to the cost of invoking a callback function associated with a specific LSM hook, which incurs roughly constant overhead. Execution refers to the cost of running the callback function, which is dependent on the specific audit (or other policy enforcement) mechanism and can also vary by the (audit) policy itself.

It is sometimes mistakenly assumed that for a given system call and a given policy on the system call, the overhead introduced by a specific LSM module would be constant. In reality, such an assumption is often an oversimplification. Consider an open system call. A number of LSM hooks, such as file_open and inode_permission, are triggered when open is called. If a new file is created because of open, additional hooks such as inode_create and inode_setattr are called when the new file's underlying inode is being created and its attributes set. Of particular interest in this example is the inode_permission hook, which is called on each directory composing the path of the file to be opened, since open must have the permission to search for the file to be opened.

To audit an open-file event, it is important to record all the permission checks (including the ones on the directories) because it reveals file access patterns. For example, in a security context, a failed inode_permission check could indicate that a compromised application attempted to scan the file system to access sensitive data. The overhead introduced by such an audit mechanism on this particular event is a function of path length. For example, assume that invoking file_open and inode_permission and running their callback functions incur the same cost C. The total overhead of a file-open event on a path of length N is $C \times (N + 1)$. Given two audit policies, P_A and P_B, such that C_A is one order of magnitude higher than C_B, the total overhead incurred by P_A is in fact *two* orders of magnitude higher than that by P_B on a path of length 10. The open system call is not the only one affected by such behavior; other system calls, such as chmod, symlink, mmap, stat, and execve have similar patterns.

Because this phenomenon can have a significant impact on the overall system performance, we analyze the call graph associated with each system call (see § 4.1) to understand LSM hook invocation patterns. We show the results for a few system calls in Table 3 (note that for readability, we do not include hooks that are called when a system call fails/errs).

5.2 sABPF overhead

sABPF's sources of overhead are fundamentally the same as those of standard LSM security modules, i.e., hooking and execution (§ 5.1). Therefore, if a standard security module and sABPF implement the same policy, they incur roughly the same total overhead, except that sABPF incurs some additional cost to traverse the cgroup hierarchy and to invoke the relevant eBPF programs (§ 3.1).

In practice, however, there exists significant differences in policy overhead between a standard security module and sABPF; sABPF offers time-saving convenience and flexibility that a standard security module is unable to provide. To run a customized in-kernel LSM module, the Linux kernel must be modified. This requires thorough testing before the deployment of the custom kernel. It is common for average users to shy away from the mere idea of deploying a kernel running heavily-customized code, especially one where said customized code interacts with the OS security framework. To mitigate those issues, standard modules are typically designed to be general-purpose. For example, an auditing module (e.g., CamFlow [51]) must be able to satisfy different auditing needs without requiring users to compile their own custom kernel. To that end, the module must evaluate *at runtime* an extensive audit policy to determine what information it should log. As a concrete example, Bates et al. [15] deploy policies to record events based on their security context as provided by SELinux. For each object involved in a given event, the audit mechanism needs to retrieve its security ID and compare it with the specified policy. While the policy is relatively simple, the cumulative effects (as discussed in § 5.1) on the policy have a significant impact on performance.

On the other hand, sABPF-based solutions take into account audit policy *at compilation time*, which significantly reduces run-time complexity and thus improves performance. Moreover, since sABPF allows users to attach programs based on cgroups, there is virtually no overhead imposed on applications running outside of the targeted cgroup. This means, for example, that if a Kubernetes pod deploys a complex audit mechanism, the other pods on the system remain unaffected.

6 PERFORMANCE EVALUATION

In this section, we evaluate sABPF performance on a bare metal machine with 16GiB of RAM and an Intel i7 CPU. In § 6.1, we analyze the cost of hook invocation on sABPF. Next, in § 6.2, we explore the performance gain from using

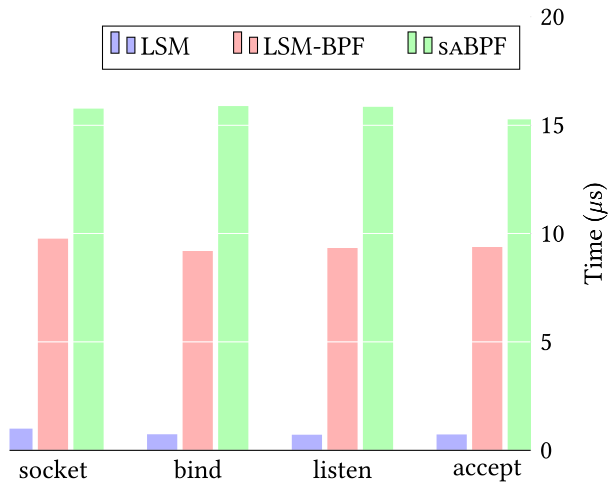

Figure 7: Overhead of the LSM, LSM-BPF, and sABPF invocation mechanisms.

sABPF rather than a state-of-the-art monitoring system that modifies the Linux kernel, when performing exactly the same functionality. Appendix A has more details on reproducing the results reported in this section.

6.1 Overhead of Namespacing

We compare the overhead associated with different mechanisms responsible for calling LSM hooks. We are interested in the following three strategies: 1) the native LSM mechanism with built-in functions (LSM); 2) LSM-BPF that attaches an eBPF function to an LSM hook (LSM-BPF); and 3) sABPF that attaches eBPF programs at the intersection of a cgroup and an LSM hook (sABPF). We use *ftrace* [9] to perform the measurement. To measure exclusively the cost of each calling mechanism, the function or program that is attached to each hook performs no operations and simply returns. We capture the overhead of four common functions associated with a UNIX server: socket, bind, listen, and accept. The overhead is associated with the following four LSM hooks: security_socket_[create/bind/listen/accept]. The results are shown in Fig. 7. We report overhead *relative* to a single baseline and focus on order-of-magnitude comparison, because ftrace (or any similar kernel instrumentation tool) can introduce additional overhead [9]. We use the overhead of native LSM on the socket_create hook to normalize experimental results.

We see in Fig. 7 that the invocation overhead is nearly constant and independent of the system call. It is roughly 10 and 15 times more costly with LSM-BPF and sABPF than with native LSM, respectively. The built-in LSM simply finds the address of the LSM function in a hook table and calls the function. The extra cost of LSM-BPF is related to the

Algorithm 1: Execute an eBPF program (simplified).

```
// disallow task core migration and preemption
migrate_disable();
rcu_read_lock();
rc = run_bpf_programs();
rcu_read_unlock();
// allow task core migration and preemption
migrate_enable();
return rc;
```

Algorithm 2: Execute an saBPF program (simplified).

```
hierarchy = get_cgroup_hierarchy(current_task,
  hook_reference);
// disallow task core migration and preemption
migrate_disable();
rcu_read_lock();
foreach cgroup in hierarchy do
    rc = run_bpf_programs();
    if rc then
        return rc;
rcu_read_unlock();
// allow task core migration and preemption
migrate_enable();
return rc;
```

cost of invoking an eBPF program. We show the simplified logic to execute an eBPF program in Alg. 1. While the overhead of executing the eBPF program itself is relatively low (close to executing a native function), handling read-copy-update (RCU) [14] synchronization primitives and manipulating scheduler migration and preemption flags accounts for the majority of the overhead.

As shown in Alg. 2, saBPF follows a similar logic, except that it incurs additional overhead when retrieving and traversing the `cgroup` hierarchy.

However, we emphasize that these relative overheads must be considered with respect to the cost of policy evaluation. As a point of comparison, SELinux’s policy evaluation cost of the `socket_create` hook is 2,000 times larger than the invocation cost of native LSM. To better understand the actual cost of running saBPF and to contextualize its overhead, we perform both micro- and macro-benchmarks in the next section.

6.2 Evaluating ProvBPF

To contextualize saBPF’s performance with a realistic workload, we perform an evaluation of ProvBPF through micro-benchmarks (§ 6.2.1) and macro-benchmarks (§ 6.2.2). We demonstrate that ProvBPF outperforms the state-of-the-art whole-system provenance solution CamFlow [51] and incurs minimal performance overhead.

We choose standard benchmarks such as `lmbench`, so that saBPF can be meaningfully compared with prior and future work. We run each benchmark on three different kernel configurations. The *vanilla* configuration runs on the unmodified mainline Linux kernel v5.11.2, which serves as our baseline. The *CamFlow* configuration uses the same kernel but additionally instrumented with CamFlow kernel patches (v0.7.2) [4]. Finally, the ProvBPF workload corresponds to the same Linux kernel but running with our eBPF-based provenance capture mechanism ProvBPF. We also ensure that ProvBPF’s and CamFlow’s configurations are equivalent.

Test Type	vanilla	CamFlow	Overhead	ProvBPF	Overhead
Process tests (in μs, the smaller the better)					
NULL call	0.30	0.32	0%	0.29	0%
NULL I/O	0.39	0.75	92%	0.54	38%
stat	1.04	3.77	263%	1.40	35%
fstat	0.52	1.40	169%	0.66	28%
open/close file	1.80	5.89	227%	2.62	46%
read file	0.40	0.73	84%	0.56	42%
write file	0.36	0.70	92%	0.53	53%
fork process	295.55	344.15	13%	317.78	8%
File and memory latency (in μs, the smaller the better)					
file create (0k)	10.31	21.20	106%	13.10	27%
file delete (0k)	11.25	23.35	108%	12.80	14%
file create (10k)	16.55	40.75	146%	20.65	25%
file delete (10k)	13.45	30.20	125%	15.55	16%
pipe latency	6.06	10.45	72%	6.55	8%
AF_UNIX latency	6.60	16.43	149%	9.72	47%

Table 4: `lmbench` results.

6.2.1 *Microbenchmark.*

We use `lmbench` [43] to measure ProvBPF’s performance overhead on raw system calls, as reported in Table 4. We show only a relevant subset of performance metrics due to space constraints, but the complete results are available online (see Appendix A).

The overhead of ProvBPF, when compared to the vanilla kernel, is relatively low. In addition to the overhead introduced by the invocation mechanism, ProvBPF also incurs the cost of building the provenance graph elements and sending them to the user-space program. It outperforms CamFlow as it is significantly streamlined. Indeed, CamFlow uses a complex set of capture policies to allow users to tailor data capture to their specific needs [51]. Evaluating the policy at runtime can be relatively costly, especially since the effects can be cumulative (§ 5). In the case of ProvBPF, policy evaluation is performed at compilation time, so that the compiled code only captures the desired events, thus significantly reducing overhead given equivalent policies.

Test Type	vanilla	CamFlow	Overhead	ProvBPF	Overhead
Execution time (in seconds, the smaller the better)					
unpack	6.52	7.70	18%	6.59	1%
build	194.26	232.01	19%	203.70	5%
4kB to 1MB file, 10 subdirectories,4k5 simultaneous transactions, 1M5 transactions					
postmark	79.50	113.00	42%	92.50	16%

Table 5: Macrobenchmark results.

Test Type	vanilla	CamFlow	Overhead	ProvBPF	Overhead
Request/Operation per second (the higher the better)					
apache httpd	14645	10682	27%	13487	8%
redis (LPOP)	2105221	1780868	15%	1894961	10%
redis (SADD)	2073489	1721367	17%	1854162	11%
redis (LPUSH)	1630446	1401497	14%	1510000	7%
redis (GET)	2360694	1928276	18%	2102901	11%
redis (SET)	1873359	1569507	16%	1690189	10%
memcache (ADD)	44122	30444	31%	41362	6%
memcache (GET)	67895	41363	39%	62167	8%
memcache (SET)	44460	30346	32%	41355	7%
memcache (APPEND)	46730	31157	33%	43215	8%
memcache (DELETE)	67761	40735	40%	61755	9%
php	690725	613296	11%	709476	0%
Execution time (in ms, the lower the better)					
pybench	1246	1298	4%	1196	0%

Table 6: Extended macrobenchmark results.

6.2.2 Macrobenchmark. We present two sets of macrobenchmarks. The first set, as shown in Table 5, measures the performance impact on a single machine when *unpack*ing and *build*ing the kernel and running the *Postmark* benchmark [39]. These are the common benchmarks used in prior provenance literature ever since Muniswamy-Reddy et al. [48] introduced the concept of system provenance. Table 6 shows the results of the second set of benchmarks focusing on a set of applications typically used to build web applications. These benchmarks are not intended to cover every possible scenario, but rather to provide meaningful points of comparison. We rely on the Phoronix Test Suite [13] to perform these benchmarks. Details on benchmark parameters and settings are available in our repository, see Appendix A.

From the first set of benchmarks (Table 5), we see that ProvBPF introduces between 1% and 16% overhead. Unpack and build workloads are computation heavy, and most of the execution time is spent in userspace. On the other hand, postmark spends a more significant portion of its execution time in system call code. As ProvBPF only adds overhead when system calls are executed, it unsurprisingly performs worse in the Postmark benchmark. In the second set of benchmarks (Table 6), we evaluate the impact of ProvBPF on applications that are often deployed through containers. ProvBPF's overhead is between 0% and 11%. In all scenarios, ProvBPF outperforms CamFlow.

We also note that ProvBPF results are in the same order of magnitude as similar whole-system provenance capture

Date	Release	Long Term Support	Changes
April 2021	5.12	No	4
February 2021	5.11	No	3
December 2020	5.10	Yes	2
October 2020	5.9	No	0
August 2020	5.8	No	4
May 2020	5.7	No	0
March 2020	5.6	No	0
January 2020	5.5	No	5
November 2019	5.4	Yes	1

Table 7: Changes made to the LSM ABI in terms of the number of interface function modified (including name changes, parameter modifications, and additions and deletions) since the latest release. We note that there is a total of 236 LSM hooks as of release 5.12.

solutions such as Hi-Fi [54] and LPM [17]. We are not able to provide direct comparison with these solutions since they were implemented for extremely outdated kernels (release 2.6.32 from 2009 for LPM [16] and release 3.2.0 from 2011 for Hi-Fi [53]); internal kernel changes make it practically impossible for us to port them to a modern kernel release.

7 DISCUSSION

Security. We are aware of a number of security issues with eBPF (e.g., CVE [6] and CVE [7]). In many known attack scenarios related to eBPF, an attacker exploits the eBPF verifier to make illegal modifications of kernel data structures, e.g., to perform privilege escalation [6]. One clear solution is to improve the verification of eBPF programs [28, 50]. While this is an important problem worthy of investigation, it is orthogonal to saBPF and therefore out of scope for this paper. We note that, to the best of our knowledge, saBPF does not introduce new attack vectors and that any improvement to eBPF security will benefit saBPF.

Layering. In this work, we focus on capturing kernel-level audit data that describes low-level system interactions. However, to fully understand application behavior, it is often useful to analyze audit information from multiple sources, preferably from different layers of abstraction. For example, layering both low-level system traces and higher-level application traces can often facilitate attack investigation by enabling forensic experts to identify, in an iterative fashion, an attack point of entry [40]. The application of such techniques is beyond the scope of this paper, but saBPF and any application built atop can be seamlessly integrated with existing layering techniques.

Maintainability. One of the key advantages in building audit tools through eBPF and by extension saBPF is that they can be heavily customized to fulfill the needs of the user.

As we previously pointed out, maintaining bespoke built-in audit tools requires the developers to, at a minimum, 1) maintain a custom kernel, 2) prepare a custom OS distribution, and 3) perform extensive testing before actual deployment. This burden is greatly alleviated using our proposed solution. The audit mechanism is neatly separated from the OS and can be built and tested independently. Furthermore, with BTF and CO-RE[49], any solution built with saBPF does not need to be built against a specific version of the kernel; it only needs to be rebuilt (and updated) when the kernel's internal LSM ABI changes, which is rare (Table 7). We note that a number of popular distributions ship only Long Term Support kernel versions, which further simplifies maintenance.

8 RELATED WORK

saBPF is designed mostly for monitoring containers in the cloud and uses two major technologies, eBPF and LSM. We discuss related work in these areas.

eBPF-based Security. In system security, one of the well-known eBPF-enabled applications is `seccomp-bpf`, which filters system calls available for user-space applications to reduce kernel attack surface [21]. `seccomp` filters use BPF programs to decide, based on the system call number and arguments, whether a given call is allowed or not. A more recent application of eBPF is LBM [62], which protects the Linux kernel from malicious peripherals such as USB, Bluetooth, and NFC. LBM places interposition hooks, through the implementation of new eBPF program types, right beneath a peripheral's protocol stack and above the peripheral's controller driver, so that it can guarantee that eBPF programs can filter all inputs from the device and all outputs from the host. LBM introduces a new filter language for peripherals to enforce programmable security policies.

LSM-BPF is still a nascent eBPF extension, emerging from Kernel Runtime Security Instrumentation (KRSI) [57]. KRSI enables privileged users to dynamically update MAC and audit policies based on the state of the computing environment. `bpfbox` [24] uses LSM-BPF to create process sandboxes through a flexible policy language. BPFContain [23] uses LSM-BPF to enforce system-wide policy to control container IPC, and file and network access. They both leverage eBPF because it is easier to maintain and further develop eBPF programs than to use out-of-tree LSMs for their particular needs. saBPF is orthogonal to these systems and focuses on leveraging eBPF at the intersection of `cgroup` and LSM hooks, mainly but not exclusively for secure auditing.

Monitoring Containers. There are a number of widely available solutions, such as Cilium [5], Grafana [10], and Nagios [11], that monitor containers, but they focus primarily on performance and/or network traffic monitoring. We design saBPF not to compete with these solutions, but to allow their functionality to be extended to secure auditing. Indeed, while these frameworks provide a wealth of information, their capture methodology does not provide strong enough guarantees in the presence of an attacker. Furthermore, saBPF enables the implementation of decentralized solutions that need not be managed by the host platform.

LSM. The LSM framework [46] was introduced nearly two decades ago to Linux for Mandatory Access Control (MAC). Two of the most popular applications of LSM are AppArmor [2] and SELinux [58]. Over the years, the LSM framework has seen its usage extended to implementing mechanisms such as the Linux Integrity Measurement Architecture [55], which enables hardware-based integrity attestation, and loadpin [20], which was developed to restrict the origin of kernel-loaded code to read-only devices in ChromeOS. LSM has also been used to implement secure auditing as previously mentioned [17, 51, 54]. These work is orthogonal to saBPF; instead, saBPF is closely related to prior work that attempted to allow namespacing and stacking of LSM modules [38, 60]. They focused on enabling containers to define their own security policy within a system-wide MAC scheme. For example, Sun et al. [60] make AppArmor namespace-aware so that each individual container can have its own policy to be enforced by the host. This is a non-trivial task involving conflict resolution alongside the security namespace hierarchy. saBPF expends on such ideas by allowing containers to provide not only their own policy, but also their own totally separate mechanisms.

9 CONCLUSION

We present saBPF, a lightweight system-level auditing framework for container-based cloud environments. saBPF is built upon the widely-used eBPF framework. It is simple to use and allows individual containers to deploy – in a decentralized manner – secure auditing tools. These tools in turn enable users to implement a wide range of security solutions on container-oriented cloud platforms, such as intrusion detection systems for individual containers. Using saBPF, we were able to re-implement a state-of-the-art audit system that provides exactly the same functionality while significantly improving performance. We open-source saBPF, welcoming the community to develop and deploy toolsets that leverage our system, in hopes of further discovering the potentials of saBPF.

A AVAILABILITY

Our implementation (§ 3) and instructions to reproduce the results presented (§ 6) are available at https://github.com/saBPF-project.

REFERENCES

[1] [n.d.]. Apche Kafka. online (accessed 29th September 2021). https://kafka.apache.org/.

[2] [n.d.]. AppArmor. online (accessed 29th September 2021). https://apparmor.net/.

[3] [n.d.]. BPF ring buffer. online (accessed 29th September 2021). https://www.kernel.org/doc/html/latest/bpf/ringbuf.html.

[4] [n.d.]. CamFlow. online (accessed 29th September 2021). https://camflow.org/.

[5] [n.d.]. Cilium. online (accessed 29th September 2021). https://cilium.io/.

[6] [n.d.]. CVE-2020-8835. online (accessed 29th September 2021). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835.

[7] [n.d.]. CVE-2021-29154. online (accessed 29th September 2021). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29154.

[8] [n.d.]. eBPF. online (accessed 29th September 2021). https://ebpf.io/.

[9] [n.d.]. ftrace documentation. online (accessed 29th September 2021). https://www.kernel.org/doc/html/v4.17/trace/ftrace.html.

[10] [n.d.]. Grafana. online (accessed 29th September 2021). https://grafana.com/.

[11] [n.d.]. Nagios. online (accessed 29th September 2021). https://www.nagios.org/.

[12] [n.d.]. Open Policy Agent. online (accessed 29th September 2021). https://www.openpolicyagent.org/.

[13] [n.d.]. Phoronix test suite. online (accessed 29th September 2021). https://www.phoronix-test-suite.com/.

[14] 2021. RCU. online (accessed 29th September 2021). https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt.

[15] Adam Bates, Kevin RB Butler, and Thomas Moyer. 2015. Take only what you need: leveraging mandatory access control policy to reduce provenance storage costs. In *Workshop on the Theory and Practice of Provenance (TaPP 15)*. USENIX.

[16] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. [n.d.]. LPM source code. online (accessed 29th September 2021). https://bitbucket.org/uf_sensei/redhat-linux-provenance-release/.

[17] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. 2015. Trustworthy whole-system provenance for the linux kernel. In *Security Symposium*. USENIX, 319–334.

[18] Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, and Satya Sahoo. 2013. *PROV-DM: The PROV Data Model*. Technical Report. W3C.

[19] Sheung Chi Chan, James Cheney, Pramod Bhatotia, Thomas Pasquier, Ashish Gehani, Hassaan Irshad, Lucian Carata, and Margo Seltzer. 2019. ProvMark: a provenance expressiveness benchmarking system. In *International Middleware Conference*. ACM/IFIP, 268–279.

[20] Jonathan Corbet. 2016. LoadPin. online (accessed 29th September 2021). https://lwn.net/Articles/682302/.

[21] Jake Edge. 2015. A seccomp overview. *Linux Weekly News* (2015).

[22] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. 2002. Runtime verification of authorization hook placement for the Linux security modules framework. In *Conference on Computer and Communications Security (CCS'02)*. ACM, 225–234.

[23] William Findlay, David Barrera, and Anil Somayaji. 2021. BPFContain: Fixing the Soft Underbelly of Container Security. *arXiv* (2021).

[24] William Findlay, Anil Somayaji, and David Barrera. 2020. bpfbox: Simple Precise Process Confinement with eBPF. In *Cloud Computing Security Workshop (CCSW)*. ACM, 91–103.

[25] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. 2017. ContainerLeaks: Emerging security threats of information leakages in container clouds. In *International Conference on Dependable Systems and Networks (DSN'17)*. IEEE/IFIP, 237–248.

[26] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *International Middleware Conference*. Springer-Verlag, 101–120.

[27] Laurent Georget, Mathieu Jaume, Frédéric Tronel, Guillaume Piolle, and Valérie Viet Triem Tong. 2017. Verifying the reliability of operating system-level information flow control systems in linux. In *International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. IEEE, 10–16.

[28] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted linux kernel extensions. In *Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, 1069–1084.

[29] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. 2020. UNICORN: Runtime Provenance-based Detector for Advanced Persistent Threats. In *Network and Distributed System Security Symposium (NDSS'20)*. Internet Society.

[30] Xueyuan Han, Thomas Pasquier, Tanvi Ranjan, Mark Goldstein, and Margo Seltzer. 2017. Frappuccino: Fault-detection through runtime analysis of provenance. In *Workshop on Hot Topics in Cloud Computing (HotCloud'17)*. USENIX.

[31] Xueyuan Han, Xiao Yu, Thomas Pasquier, Ding Li, Junghwan Rhee, James Mickens, Margo Seltzer, and Haifeng Chen. 2021. SIGL: Securing Software Installations Through Deep Graph Learning. In *Security Symposium*. USENIX.

[32] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. 2018. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium (NDSS'18)*.

[33] Wajih Ul Hassan, Adam Bates, and Daniel Marino. 2020. Tactical Provenance Analysis for Endpoint Detection and Response Systems. In *Symposium on Security and Privacy (S&P'20)*. IEEE.

[34] Wajih Ul Hassan, Mohammad Ali Noureddine, Pubali Datta, and Adam Bates. 2020. OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis. In *Network and Distributed System Security Symposium*. Internet Society.

[35] Tejun Heo. [n.d.]. Control Group v2. online (accessed 29th September 2021). https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.

[36] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. 2004. Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM Transactions on Information and System Security (TISSEC)* 7, 2 (2004), 175–205.

[37] Hai Jin, Zhi Li, Deqing Zou, and Bin Yuan. 2019. Dseom: A framework for dynamic security evaluation and optimization of MTD in container-based cloud. *IEEE Transactions on Dependable and Secure Computing* (2019).

[38] John Johansen and Casey Schaufler. 2017. Namespacing and Stacking the LSM. In *Linux Plumbers Conference*.

[39] Jeffrey Katcher. 1997. *Postmark: A new file system benchmark*. Technical Report. Technical Report TR3022, Network Appliance.

[40] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Network and Distributed System Security Symposium (NDSS'13)*. Internet Society.

[41] Fucheng Liu, Yu Wen, Dongxue Zhang, Xihe Jiang, Xinyu Xing, and Dan Meng. 2019. Log2vec: a heterogeneous graph embedding based approach for detecting cyber threats within enterprise. In *Conference on Computer and Communications Security (CCS'19)*. ACM, 1777–1794.

[42] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. 2016. Fast memory-efficient anomaly detection in streaming heterogeneous

graphs. In *International Conference on Knowledge Discovery and Data Mining (KDD'16)*. ACM, 1035–1044.

[43] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis. In *Annual Technical Conference (ATC'96)*. USENIX, 279–294.

[44] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and V. N. Venkatakrishnan. 2019. Poirot: Aligning Attack Behavior with Kernel Audit Records for Cyber Threat Hunting. In *Conference on Computer and Communications Security (CCS'19)*. ACM.

[45] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, Ramachandran Sekar, and VN Venkatakrishnan. 2019. Holmes: real-time apt detection through correlation of suspicious information flows. In *Symposium on Security and Privacy (S&P'19)*. IEEE, 1137–1152.

[46] James Morris, Stephen Smalley, and Greg Kroah-Hartman. 2002. Linux Security Modules: General Security Support for the Linux Kernel. In *Security Symposium*. USENIX.

[47] Thomas Moyer and Vijay Gadepally. 2016. High-throughput ingest of data provenance records into Accumulo. In *High Performance Extreme Computing Conference (HPEC'16)*. IEEE, 1–6.

[48] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-aware Storage Systems. In *Annual Technical Conference (ATC'06)*. USENIX, 43–56.

[49] Andrii Nakryiko. [n.d.]. BPF Portability and CO-RE. online (accessed 29th September 2021). https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html.

[50] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX, 41–61.

[51] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical Whole-System Provenance Capture. In *Symposium on Cloud Computing (SoCC'17)*. ACM.

[52] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. 2018. Runtime Analysis of Whole-System Provenance. In *Conference on Computer and Communications Security (CCS'18)*. ACM.

[53] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. [n.d.]. Hi-Fi source code. online (accessed 29th September 2021). https://github.com/djpohly/linux.

[54] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: Collecting High-fidelity Whole-system Provenance. In *Annual Computer Security Applications Conference (ACSAC'12)*. ACM, 259–268.

[55] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Security Symposium*, Vol. 13. USENIX, 223–238.

[56] Z Cliffe Schreuders, Tanya McGill, and Christian Payne. 2011. Empowering end users to confine their own applications: The results of a usability study comparing SELinux, AppArmor, and FBAC-LSM. *ACM Transactions on Information and System Security (TISSEC)* 14, 2 (2011), 1–28.

[57] KP Singh. 2019. Kernel Runtime Security Instrumentation. online (accessed 29th September 2021). https://lwn.net/Articles/798918/.

[58] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* 1, 43 (2001), 139.

[59] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *European Conference on Computer Systems (EuroSys'07)*. ACM, 275–287.

[60] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security namespace: making linux security frameworks available to containers. In *Security Symposium*. USENIX.

[61] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. 2018. NodeMerge: Template Based Efficient Data Reduction For Big-Data Causality Analysis. In *Conference on Computer and Communications Security (CCS'18)*. ACM, 1324–1337.

[62] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Peter C Johnson, and Kevin RB Butler. 2019. LBM: a security framework for peripherals within the linux kernel. In *Symposium on Security and Privacy (S&P'19)*. IEEE, 967–984.

[63] Kennedy A Torkura, Muhammad IH Sukmana, and Christoph Meinel. 2017. Integrating continuous security assessments in microservices and cloud native applications. In *International Conference on Utility and Cloud Computing (UCC'17)*. IEEE/ACM, 171–180.

[64] Veritis. 2019. State of Containers Report 2019: 'Security' Remains A Challenge! online (accessed 29th September 2021). https://www.veritis.com/blog/state-of-containers-report-2019-security-remains-a-challenge/.

[65] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A. Gunter, and Haifeng Chen. 2020. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In *Network and Distributed System Security (NDSS'20)*. Internet Society.

[66] Robert NM Watson. 2007. Exploiting Concurrency Vulnerabilities in System Call Wrappers. *Workshop on Offensive Technologies (WOOT'07)* 7 (2007), 1–8.

[67] Robert NM Watson. 2013. A decade of OS access-control extensibility. *ACM Queue* 11, 1 (2013), 20–41.

[68] Wenhui Zhang, Peng Liu, and Trent Jaeger. 2021. Analyzing the Overhead of File Protection by Linux Security Modules. In *Asia Conference on Computer and Communications Security (AsiaCCS'21)*. ACM, 393–406.