# SysFlow: Towards a Programmable Zero Trust Framework for System Security

Sungmin Hong[1*], Lei Xu[1*], Jianwei Huang[1], Hongda Li[2], Hongxin Hu[2], Guofei Gu[1], *Fellow, IEEE*

***Abstract*—Zero Trust, as an emerging trend of cybersecurity paradigms in modern infrastructure (e.g., enterprise, cloud, edge, IoT, and 5G), is moving security defenses from static and perimeter-based control systems to focus on users and resources with no assumption of implicit trust. However, the current Zero Trust Architecture (ZTA) mainly focuses on the network security and lacks in-depth considerations on system-level security policies and abstractions, which leaves the realization of the principle incomplete. To bridge the gap, we propose an innovative *programmable* system security framework called SYSFLOW to enable unified, dynamic, and fine-grained Zero Trust security control for system resources. SYSFLOW introduces a novel *system flow* abstraction to model *system activities* across the entire infrastructure, and provides a system-level data plane and control plane separation and abstraction. The new logically centralized controller accommodates a unified *programmable* Policy Decision Point (PDP) that acquires a holistic view of system behaviors for controlling system resource accesses by translating programmable security policies into system flow rules. The SYSFLOW data plane, acting as Policy Enforcement Point (PEP), enforces translated system flow rules, which can be updated dynamically and facilitate fine-grained responsive actions. Our extensive evaluations demonstrate the effectiveness and scalability of SYSFLOW, which addresses the security issues in various scenarios with a minor performance overhead.**

## I. INTRODUCTION

The proliferation of personal mobile devices, cloud-native apps/services, containers, and the Internet of Things (IoT) has trespassed traditional security boundaries. Modern enterprise security must evolve to manage the complex task of handling continuously changing risks from various locations in a fine-grained manner. That said, not only are enterprises responsible for catering secure user access to critical enterprise resources regardless of locations and devices but also they provide microscopic security measures to protect every sensitive resource even in a single host system.

Unfortunately, existing perimeter-based network security has been a criticized part of enterprise security. Considering the growth of highly dynamic mobile applications, cloud computing and containers, IoT devices, etc., the traditional security boundary becomes blurred out. Furthermore, narrowing down to host-level system security, existing security practices are mainly coarse-grained and static. Security policy enforced in a host system cannot easily handle the dynamics of modern applications in a flexible manner. For instance, dynamic migration/scale-out of virtual machines and containers spanning across multiple hosts cannot be easily dealt with the current security tools such as mandatory access control (MAC), host-based intrusion detection system (HIDS), and antivirus (AV) due to the lack of global visibility.

To address these emerging security risks, the zero trust (ZT) [43] security concept has recently been proposed to focus on resource protection with a guiding principle that trust is never granted implicitly but must be continually validated. Under this principle, resources must be secured from malicious subjects (users, applications, and other non-human entities that request information from resources) with finer-grained perimeterization. However, different from the mainstream Zero Trust Architecture (ZTA) [4], [8], [16], [23] focusing on network security, in this paper, we scrutinize the issues of system security to motivate ourselves to come up with a new system security framework.

Modern computing facilitates microservice creation, termination, and migration regardless of physical locations, thereby trespassing the traditional perimeter dynamically. Suppose that a business microservice (e.g., eShop) is dynamically migrated to a host system (e.g., a VM in the cloud) where other containers are running together as depicted in Figure 1. Once the attacker passes the network security perimeter provided by the existing ZTA, the next level of security must be system-level perimeters. By leveraging container vulnerabilities (e.g., CVE-2019-14271 [42]), the attacker may breach the first line of a system-level perimeter that restricts inter-container views based on configurations, i.e., namespace isolation. Even though either a container provides limited functionality and permissions in an isolated space or a microservice is updated with a new logic/patch, the vulnerabilities of the container environment open the gate to enable malicious cross-access between the container and the system. As a result, all resources in the host system and other containers are maliciously accessed by the compromised privileged container. It fails to realize the ZT principle since there is no easy way to continually maintain security contexts and analyze/evaluate the risks of access either between a container and a system, or across systems in infrastructure with the existing standalone system security tools [3], [17].

In this paper, we investigate the Zero Trust principle for programmable *system security* to protect sensitive resources in a host system constantly. We extend ZT principle to the system level to control, monitor, and verify accesses at all times and gain a global view of subjects and resources across networks. Also, to better cope with the dynamics, we provide

* The first two authors contribute equally to the paper.

[1]S. Hong, L. Xu, J. Huang, and G. Gu (correspondence author) are with the SUCCESS Lab, Dep. of Computer Science and Engineering, Texas A&M University, College Station, TX 77843 USA (Email: ghitsh@tamu.edu, lexu@paloaltonetworks.com, hjw@tamu.edu, guofei@cse.tamu.edu)

[2]H. Li and H. Hu are with the Dep. of Computer Science and Engineering, University at Buffalo, Buffalo, NY 14260 (Email: hli@paloaltonetworks.com, hongxih@clemson.edu)

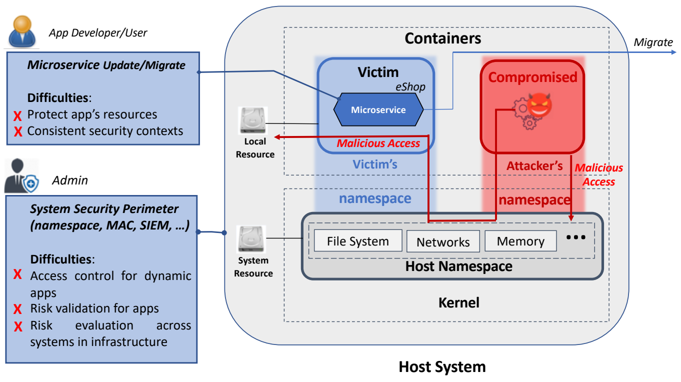

Fig. 1. Motivating Example

the programmability of system security. As a security posture of a microservice at one location is ephemeral, the framework should be able to dynamically identify the changes of contexts and move/install the ZT security policy accordingly. Furthermore, security logic to verify the risks of malicious access with flexible system-level visibility with contexts can be programmed through predefined risk verification algorithms or user-defined security algorithms provided by admins/developers.

However, we encounter several research challenges in designing our security framework. Regarding the abstraction of system activities and modeling of system capabilities, a generalized, global definition should be necessary among different host platforms from a compatibility perspective. How to model heterogeneous subjects, such as process and container as a common abstraction would alleviate confusion to security application developers. To enable *high-level security programmability* for Zero Trust at an infrastructure scale, the framework should support expressive APIs that are not restricted to specific applications, domains, or languages. Moreover, the security logic should be dynamically enforced and updated in response to the dynamics but the realization is non-trivial to maintain consistency. The visibility is of critical importance to precisely monitor the behavior of the systems; however, how to provide admins/developers with a system-level visibility flexibly with sufficient contexts at runtime is challenging as well. Last but not least, how to achieve minor performance overhead is also a significant engineering challenge.

To address the aforementioned challenges, we propose a novel system security framework called SYSFLOW for *unified*, *infrastructure-wide*, *dynamic*, and *fine-grained* flow-level *programmable* security control of system resources, with the emerging Zero Trust (ZT) principle in mind. SYSFLOW is a new programmable Zero Trust system security framework which enables admins/developers to easily write a security application using SYSFLOW APIs to realize some key Zero Trust features such as micro-segmentation (which provides finer-grained, programmable access control and isolation of system resources) and risk awareness (which continuously maintains and evaluates risks of accesses), as well as general system security functions. The primary goal of this work is to revisit the system security policies and abstractions with respect to modern computing infrastructure. Sitting in the heart of SYSFLOW is a general system activity abstraction over existing system security capabilities. In particular, SYSFLOW introduces a flow-based model, namely *system flow*, to abstract system activities. Inspired by the programmable network flow model [15] in Software Defined Networking (SDN), a system flow consists of 3 tuples (source, destination, and operation) to generically and formally reason about the state of diverse system activities. In addition, based on the system flow model, *system flow rules* are introduced to represent system security intents.

The flow-based model, introduced by SYSFLOW, provides a system-level data plane and control plane separation and abstraction. As a result, SYSFLOW embraces a two-layer architecture, which includes two major components, i.e., SYSFLOW Data Plane (SDP) and SYSFLOW Controller (SC). In the SYSFLOW control plane, the logically centralized SC acquires a *holistic* view of security contexts from the low-level abstraction of system activities and provides a *unified* programming abstraction, even across the entire infrastructure, to facilitate the flexible implementation and deployment of diverse SYSFLOW security applications based on system flows. SDP automatically enforces system flow rules to enable *fine-grained* responsive security actions, and to *dynamically* update security intents (in the form of system flow rules) according to the change of contexts. In particular, the flow-based model treats subjects (users, applications, and other non-human entities that request information from resources) as a common entity for generic programmability. Also, it can enable dynamic reconfiguration of security policies through our reactive and proactive programming model.

The key contributions of this paper are as follows:

- We introduce a unified programming abstraction for host systems, namely *system flow*, which can facilitate the specification and enforcement of diverse system security intents for general-purpose system security.
- We design and implement SYSFLOW, a system security development framework for Zero Trust, to provide admins/developers with expressive programming interfaces to easily realize micro-segmentation and risk awareness.
- SYSFLOW provides global, system-level visibility and logging capability for system activities at runtime, which can leave a door open for admins/developers to program any useful security algorithms that fully leverage infrastructure-wide visibility.
- Our extensive evaluations show that SYSFLOW is useful to develop various types of system security apps in practice and only incurs minor performance overhead.
- We release the source code [20] to benefit future ZT systems and security research on top of SYSFLOW as a complementary framework to network-based ZTA.

## II. PROBLEM STATEMENT

To realize Zero Trust for system security, resource access requests and behaviors of subjects should be evaluated continuously in real-time over the actions inside the system. To do so, a basic step would be to find how to abstract system information/behavior and identify an interesting common set of security primitives/functions at a flexible level of granularity. On one hand, existing system call/log-based solutions are too fine-grained and expensive with extensive raw and low-level system information for Zero Trust, which is costly for the centralized control plane to manage. Furthermore, handling such unstructured information makes the security system complicated and less compatible with heterogeneous systems/platforms. On the other hand, if the abstraction is too coarse-grained, we may lose granularity and flexibility for Zero Trust applications. For example, AppArmor [3] abstracts a system in a less complex and easier way using file paths instead of the labels of SELinux [17] (an instantiation of Flask [45]) whose policy complexity is an often criticized drawback [36]. It would be more usable, practical to propose a balanced abstraction by modeling system information/behavior not only with an interesting common set of security primitives/functions for Zero Trust but also with capabilities that developers can customize.

The Zero Trust Architecture should allow developers to enforce security policies to cope with a huge amount of system events with low-level system details and oftentimes change security algorithms in response to security threats. However, existing MAC approaches (e.g., SELinux [36], AppArmor [3]) are limited to configure access control in a local view where a security server (policy decision point) is merely a kernel subsystem. Not only such access control approaches, per se, cannot continuously verify the risk of access requests without additional security tools but also adding any new security logic/functions is not supported in general. Moreover, the dynamics of hosts/containers require configurations and labeling (SELinux) to change dynamically, which is challenging to keep consistent with a local security server. In addition, host-based security tools, e.g., HIDS, Anti Virus (AV), or DIFT systems [35], [54] also suffer from laborious, manual efforts in tracking context and reconfiguration which makes it difficult to cope with dynamic, ephemeral characteristics of cloud-native computing.

More visibility into the system can typically help detect the signs of security problems. It is common to use SIEM systems (that log, send, and analyze at an infrastructure scale) for system security in most Zero Trust Architecture [43]. However, we may face different challenges in terms of security and performance. Many practices in enterprises reveal that security analytic tools require a comprehensive collection of raw system events from all host systems using audit logs sent to a remote server. 3rd-party tools, then, model the raw information to their definitions and render a security analysis, e.g., anomaly detection, causality/correlation analysis, etc. However, not only this ubiquitous monitoring is not a real-time tool, but also it introduces significant pressure to storage systems both in a host and a log server. Also, the capabilities of viewing the system internals from the existing tools are limited to the general system activities between system objects (process) and resources. For security applications, it is significantly useful to precisely investigate which process is running in what containers and micro-services by which user to handle the dynamics.

Last but not least, realizing system security for Zero Trust requires solving the performance degradation. As introduced in ZTA [43], most mainstream ZTAs [4], [8], [16], [23] follow the centralized policy-decision-point architecture by decoupling data and control plane to effectively control authentication and risk verification. However, this architecture would be an inherent hurdle when applying to Zero Trust system security where a large volume of extra system information should be monitored and exchanged. In this paper, we plan to investigate and answer the following research challenges.

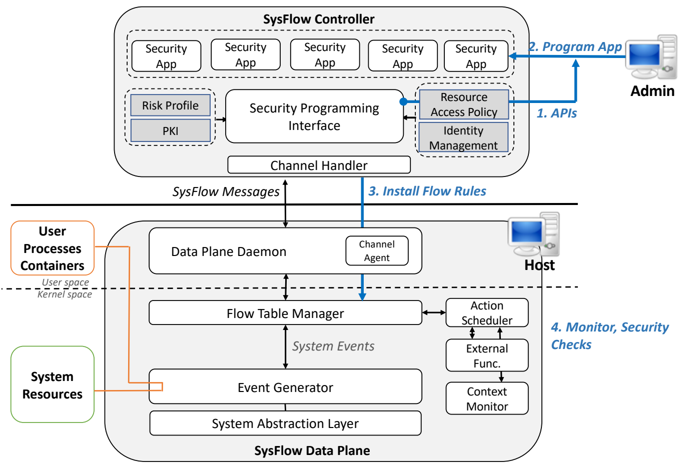

Fig. 2. SysFlow components and workflow.

- **(C1)** How to *abstract* system activities and model security capabilities for unified programmability in a generalized way? (§ IV)
- **(C2)** How to enable *unified high-level security programmability* and handle the dynamics accordingly? (§ V)
- **(C3)** How to provide flexible visibility in order to achieve context-aware Zero Trust control? (§ V)
- **(C4)** How to achieve *minor* performance overhead on a host system? (§ VI)

## III. System Overview and Threat Model

**System Architecture.** As depicted in Figure 2, SysFlow embraces a two-layer programmable design that includes SysFlow *Data Plane* (SDP) and SysFlow *Controller* (SC) in line with ZTA [43]. The control plane, as Policy Decision Point (PDP), is used by various infrastructure components to maintain assets; judge, grant, or deny access to resources; and perform any necessary operations to set up communication/access paths between resources. The data plane acts as Policy Enforcement Point (PEP) which is responsible for enabling, monitoring, and eventually terminating access between a subject and an enterprise resource.

SDP runs in target host systems. SDP Daemon resides in the user space of the system that is used to intercept communications between SC and Flow Table Manager in the kernel. It talks with SC by using the SysFlow control messages (detailed in Section VII and interacts with Flow Table Manager to manipulate flow tables accordingly. System Abstraction Layer (SAL) abstracts low-level system activities to common definitions to support compatibility among different operating systems. Event Generator generates system events based on SAL and further inputs those system events to Flow Table Manager. Flow Table Manager maintains system flow rules (in a flow table) to match system events and trigger Action Scheduler to enforce corresponding actions to control system activities. Context Monitor supports visibility with contexts by not only continuously monitoring behavior and the context of processes/containers based on the flow model but also capturing application-level context/profile to associate an application and a user over the access to system resources. External Security Function offloads part of/the whole security functions into SDP to reduce the interactions as well as enabling diverse security functions to place in the data plane for the immediate, expressive response. Any type of security functions can be installed by security applications on the fly, e.g., a customized system call pattern analysis, complex container defense code, deep packet inspection, and redirection to other machines for the in-depth analysis.

SC works as a logically centralized control and management nexus that serves interfaces to collect context information from all host systems running SDP and install system flow rules accordingly. The controller provides a unified high-level programming abstraction and interface to *facilitate the development of security applications to enforce diverse security intents* (more design details are described in Section V). Also, the controller accommodates several extensible supporting components to help admins configure a core security policy, such as resource access policy (pre-defined access control rules by admins), risk profile, identity management (host/process/-container/user identity database and management), public key infrastructure (key storage for a secure connection between SC and SDP and for external functions).

**General Workflow.** To show how SysFlow works (Figure 2), suppose that admins write a security app, *File Reflector*, for cyber deception to divert attackers away from sensitive data. Based on ① SysFlow APIs, ② the admins write the *File Reflector* app (written in Java in our implementation). The app contains flow rules with actions of *redirect* to divert access to a honey file and *log* to observe further system activities, as well as a risk profile to validate the risk score of the process. ③ SC installs a set of flow rules converted from the app into SDP. Then, SysFlow monitors the system events specified in the installed flow rules. ④ When a suspicious process determined by the risk profile attempts to access sensitive resources, the *redirect* action will be executed to redirect file operations to the honey file. To investigate further system activities, the flow rules with the action, *log*, will inform the app of system activities from the process. The details of the SysFlow flow model and APIs are elaborated in Section IV and V.

**Threat Model.** Similar to prior system security approaches [24], [31], [32], [50], we first assume that the kernel, in which SDP is running, the communication channels, and the server running SC are trusted computing base (TCB). Users, applications, and containers are not trusted as in the Zero Trust model. We consider that an adversary may attempt to compromise the availability or privacy of the system resources protected by SysFlow. In this case, the adversary (in the user space), for instance, may install malware/ransomware, exploit running processes, or launch denial-of-service (DoS) attacks.

In addition, we make the following assumptions. First, attacks will happen only after the initiation of SDP and the controller. Second, attacks based on hardware and side/covert channels of shared system resources are beyond the scope of the paper. Third, SysFlow can leverage state-of-the-art integrity-checking mechanisms [33], [44] to determine if there are any compromises against SysFlow components, especially SDP Daemon in the user space. Emerging hardware-assisted protection mechanisms such as SGX [26] could also be used to further protect user-space SysFlow components. Fourth, changing task_struct to subvert the subject identity is assumed to be protected [41]. Lastly, we assume that system admins and security experts who write security applications are trusted.

## IV. System Flow Abstraction

### A. System Security Abstraction

Abstracting the system security is a fundamental step to provide a higher-level security function atop. Although we design a similar system abstraction to AppArmor based on LSM (Linux Security Module), our design choices are clearly different from AppArmor which focuses only on access control with binary actions (allow/deny). Given the Zero Trust architecture that incorporates existing system security tools to manage the entire infrastructure, the challenges may include the lack of balance on complexity in *expressibility*, *extensibility* among heterogeneous host systems, and *dynamic programmability* for security functions. The goal of our system security abstraction should be easy to express, yet contain rich *security functionalities* than the prior security tools (i.e., MAC). The host systems and security services should neither halt nor be delayed due to the resource labeling or remote log-based analysis. System security framework should be able to extend the abstraction and security functionalities. Admins/security experts should dynamically program their security logic/function.

To this end (**C1**), SysFlow introduces a flow-based model for system activity abstraction, called system flow. Inspired by SDN, we define a system flow as an interaction between subjects (e.g., processes) and observation/access points (system objects), e.g., file, memory, pipe, and socket, in the system during a certain time interval. SysFlow constructs a flow table with a flow match in an entry characterized by src/dst system objects/resource, system operations, and metadata. Note that, based on this flow definition, SysFlow provides system-oriented aggregated views on the system activities, just as SDN supplies common low-level network views and building blocks via APIs upon which admins/security experts can write arbitrary controller apps (e.g., various security functions such as stateful firewall and DDoS detection).

**System Event.** In an infrastructure, host systems may include a variety of critical activities, which should be monitored or controlled according to different security policies. Hence, we define a general concept of *system events* to model such critical system activities for Zero Trust. In this paper, we particularly adopt system events to cover interactions between programs/processes and different system resources (e.g., file, Notations: Integer *n*, Wildcard *

| | | |
|---|---|---|
| **Match** | *ma* | ::= <*src, dst, type*> |
| Source ID | *src* | ::= *id* \| * |
| Destination ID | *dst* | ::= *id* \| * |
| Resource ID | *rid* | ::= *id* \| * |
| Operation Type | *type* | ::= *n* |
| ID | *id* | ::= $\{n_1, n_2, \ldots, n_k\}$ |
| **Action** | act | ::= pa \| (act \| act) \| (act >> act) |
| Primitive Actions | *pa* | ::= allow \| deny \| report\| message \| log \| encode(tag) \| decode(tag) \| redirect ($dst_1$, $dst_2$) \| quarantine ($pid_1$, $rid_2$) \| external(*id*) |
| **Priorities** | *pri* | ::= *n* |
| **System Flow Rule** | *rule* | ::= <*ma, act, pri*> |

Fig. 3. Syntax of system flow rule.

socket, and IPC) to model the continuous observation of subjects' resource access request and behaviors. The key difference from the existing work (i.e., MAC approaches) is that the system events are extensible through Event Generator as detailed in Section IV-B.

**System Flow Rule.** SYSFLOW introduces *system flow rules* to model system security capabilities upon a sequence of system events. A system flow rule is formally defined by the syntax listed in Figure 3. System flow rules are used to capture system security intents, which include *match*, *action*, and *priority*. A *match* is a predicate to match system events that have the same attributes, i.e., *source*, *destination*, or *type*. The source of a system flow is the initiator of the flow, such as processes and users. The destination of a system flow is the receiver of the flow, such as files, memories, and sockets. The type of system flow is used to classify the different interactions between system applications and resources, e.g., writing a file. Note that a system flow can be used to represent an exact system event or a group of system events with the same pattern by using a wildcard notation (*). For example, a system flow can be specified as $\{src: *,\ dst: file_1,\ type: file\_op\_write\}$ to match system events representing any process writes to $file_1$. A system flow rule uses a list of *primitive actions* to specify how the system events should be processed. An innovation over the conventional LSM-based MAC approaches is that SYSFLOW provides several useful security primitives beyond the capabilities of LSM as well as the extensibility of the actions. Whereby, security functionalities can be easily programmed/extended with the flow model, which is different from single-purpose security tools such as MAC and SIEM. For example, *redirect* aims to change the system flow/operations to a new location (e.g., system objects and resources) as briefly introduced in *File Reflector*. However, the enforcement of *redirect* action is challenging to design only with the existing hooks of LSM. To this end, we place an additional hook to handle the *redirect* action. Since a process in the user space manipulates a file through a file descriptor, we place a hook (*fd_bind*) before the file descriptor is bound to a specific inode. By invoking the *fd_bind* hook, we can enforce the *redirect* action by replacing the inode of the original file with the inode of any other files. *encode and decode* are designed to push/check a contextual tag into/from network packets, which can be used to enforce cross-host information flow tracking to protect from data leakage. We design this action with *netfilter* due to the limitation of LSM hooks that do not allow packet modification. Moreover, we design the *external* action to embed any security logic written by security application developers to run close to resources in SDP. However, the challenge is that running codes in a kernel module (LSM) requires loading the kernel module and is not safe to run as sand-boxed programs. To address this problem, any security logic is designed to run in an eBPF (Extended Berkeley Packet Filter) VM without changing kernel code or loading a kernel module.

To address the possible conflicts between different flow rules, an integer-based priority is used to disambiguate rules with overlapping patterns. If a system event matches multiple system flow rules, only the highest-priority rule will be applied.

### B. Flow-based Programming Abstraction

In this section, we detail how SYSFLOW supports flexible system event generation and how security application developers control low-level system objects on their applications.

**System Event Generator.** The role of the Event Generator component is to generate system events by intercepting system-level activities (e.g., system calls) from hooks in host systems. It also interprets the semantics of system events by parsing parameters from those hooks if necessary. Different from the existing work, Event Generator provides an interface, called *sysflow_generate_event*, to allow admins to generate their own system events to easily extend security functionalities and expressibility with the flow model. Event Generator further inputs system events to Flow Table Manager to reference system flow rules in the flow table and enforce corresponding security actions.

**Resolving Resource Identifier.** In many cases, the security application developers encounter semantic gaps for system objects. For example, a security application developer may not know the identifiers (i.e., *UUID* and *inode number*) of personal tax files in the file system of the victim host when they want to write security apps to prevent the exfiltration attacks. Instead, they may be aware of the file name and possibly the path. To bridge the semantic gap, Flow Table Manager enables an identifier binding and resolution service for system objects.

At runtime, Flow Table Manager maintains the profile table for system objects. For example, Flow Table Manager will keep the name of a process in addition to the process identifier. When receiving the requests to update flow rules that include attributes of system objects instead of identifiers, the Flow Table Manager will refer to the profile table to retrieve the identifier of the system object. In this case, we will install multiple flow rules if the identifier is not unique. Besides, the binding service will also monitor the change of the mapping from non-identifier profiles to the identifier (e.g., from name to *UUID* and *inode number* for a file) at runtime and update the corresponding flow rule accordingly.

## V. SYSTEM SECURITY PROGRAMMABILITY

Our approach is to provide developers with high-level and unified programmability for Zero Trust as well as **generic** system security applications. By leveraging the *decoupled* architecture, security applications remain less dependent on the internals of host systems. Integrating separate system security functionalities such as access control, isolation, monitoring, and behavior analysis is a non-trivial task in practice. SYSFLOW integrates multiple system security functionalities into a single framework as a unified tool, by allowing users to use Zero Trust APIs to develop their security applications with their security functions/algorithms with less hassle.

In this section, we describe how we design SYSFLOW to provide programmability with finer-grained visibility for Zero Trust system security to address (**C2**) and (**C3**).

### *A. Programmable Resource Control with Micro-segmentation*

Micro-segmentation is a term typically used as a method of creating logical/virtual security zones in *network environments* to isolate workloads from one to another and secure them individually. When we apply this concept to system security, it is challenging to realize it. As illustrated in our motivating example (Figure 1), The basic sandboxing mehcanism of the container (i.e., Docker) is Linux namespaces. However, namespaces are tied to resources of the host system that cannot be isolated since file systems (e.g., *cgroups* and *sysfs*) are shared with the host system. Thereby, a compromised host can access the sensitive resources that belong to other containers and system resources through vulnerabilities. To restrict access to resources among containers, we might utilize a separate MAC profile (e.g., AppArmor) per container by placing each one in a separate security context. However, the profile requires the MAC to restart due to merging into a global profile and starting with the new one. In addition, the profile is restricted to a single host, thereby neither being able to build multiple logical segments nor being installed/updated dynamically upon container migration. Moreover, upon changes/patches/fixes of app/business logic/container/microservice, the existing security contexts (e.g., interaction related to resource access, behaviors of processes, etc.) cannot be easily maintained with other security tools. These highlight why simple updates/patches of either problematic containers or security profiles for a single container are difficult to apply to Zero Trust since security contexts cannot be consistently, persistently maintained/shared for infrastructure.

To address this problem, we design useful, easy-to-use micro-segmentation APIs to provide security applications with convenient programmability as shown in Table I. The identifier of the micro-segmentation APIs provides admins/security experts to build their own logically-separate security zone either by nesting other micro-segmentation or by including new flow rules, for example, per-user/-container/-system/micro-service profiles. The APIs, then, are converted into a list of flow rules for the system to be installed dynamically without restarting the framework. Furthermore, the key innovation over the traditional ossified security tools designed for a dedicated purpose is that different security applications are easily assigned/interfaced to each micro-segmentation. For instance, per-container and -system micro-segmentation may be applied to a default common set of access control security application but per-micro-service and -user micro-segmentation can be

TABLE I
LIST OF MICRO-SEGMENTATION APIS.

| API | Descriptions |
|---|---|
| ms_create(ms_id) | Create a new micro-segmentation. |
| ms_kill(ms_id) | Delete a micro-segmentation. |
| ms_alloc(ms_id, flow) | Allocate system flow into micro-segmentation. |
| ms_free(ms_id, flow) | Remove system flow from micro-segmentation. |
| ms_acl(ms_id, profile) | Configure risk profile for a micro-segmentation. |

applied to a system firewall and HIDS security application for specific security purposes. Nevertheless, we face the following technical challenge when we design the micro-segmentation: how to handle the micro-segmentation with security contexts upon the dynamic migration?

**Dynamic Policy Programming.** To support a dynamic policy, SYSFLOW provides both *reactive* and *proactive* programming paradigms.

In a reactive manner, SC first installs monitoring system flow rules into the data plane, which will report a set of system events (as contexts) to the controller. Then, the controller installs corresponding responsive flow rules to react/respond to the contexts from the reported system events.

The advantage of our reactive programming is that SYSFLOW can support tracking the instantiation of containers dynamically. Unlike the existing security tools that should constantly keep the consistency from orchestrators for the dynamics of containers and manually update security policies accordingly, SYSFLOW reactively collects container information and updates security policies accordingly. For example, suppose a flow rule is watching if *nginx* component of an eShop micro-service attempts to take any action, e.g., {*nginx, *, *, report, priority*}. Once the flow rule hits, SYSFLOW will inform the security application. Then, the application will request SDP to retrieve contexts of the process such as container ID, micro-service name, creation time, etc. Now that the application recognizes the container is dynamically instantiated and it can further install a security policy, e.g., denying attempts to access tax files, e.g., {*nginx, tax_files, file_open, deny, priority*}

Reactive flow installation may add an extra latency during the communications between the data plane and the controller. Instead, SYSFLOW also provides a proactive flow installation to allow a developer to offload system event processing logic into the SYSFLOW external security function in the SDP, as detailed in Section VI-B.

### *B. Risk-aware Security Response*

From the motivating example, a malicious user of the *eShop* microservice who is authenticated with valid credentials may have access to other user's resources through vulnerabilities in the same container. The compromised container can also break into the host system resources with a root privilege through container vulnerabilities (e.g., *runC*, a CLI tool that runs for each container). It is challenging for the existing MAC approaches to defend against such attacks because no access control rule/action to investigate the system activities caused by the authenticated microservice user. Also, not only simply denying *runC* would stop low-level container operations but the lack of continuous assessment of a risk for Zero Trust cannot handle the problem. The aforementioned problems highlight why the existing per-container isolation with limited functionality/permissions cannot be assumed to be protected reliably, further allowing malicious cross-access due to the hole/bypass of container security perimeter. Note that although the example emphasizes on container security, it applies to generic system security in the context of risk-awareness. SYSFLOW is a generic system security framework to address such problems that isolation-only approaches suffer from. Here, we summarize three technical challenges to address the problem: (1) how to provide programming interfaces to evaluate risk?, (2) how to create meaningful security responses for a further investigation in lieu of making an allow/deny action?, (3) how to provide finer-grained security contexts with flexible monitoring points for the precise risk assessment?.

For the first challenge, to dynamically inspect malicious access requests and make a reasonable decision based on a confidence/risk level, SYSFLOW provides risk profile templates as APIs for different schemes to support risk awareness, including privilege-based risk profile, context-based risk profile, and scoring-threshold-based risk profile. On top of that, SYSFLOW enables security applications to write customized risk logic/algorithm. The privilege-based risk profile is to directly grant an action for resource accesses, i.e., allow or deny. The privilege-based risk profile is designed for two reasons. First, it provides the compatibility to existing rules, which means admins can easily migrate from their existing solutions to SYSFLOW. Second, it allows admins to define specific rules that cannot be covered by other security applications in SYSFLOW. Also, the context-based risk profile is to decide access control actions based on the attributes of the request, e.g., name, create time, and visit history. In addition, a developer can choose specific scoring algorithms (e.g., EigenTrust [47]) to rate each access; if the request is over the threshold, then the request can be granted. Such the general risk profile may not be applicable to all the cases. Specifically, when malicious behaviors are detected by any security applications, the scoring-threshold-based is more flexible than the previous two since the threshold can be dynamically modified based on the global risk level.

To address the second challenge, SYSFLOW facilitates the programmable *action*. Suppose that the malicious user attempts to read a legitimate user's sensitive file and send it to a remote location in a network from the motivating example. When a security application detects an abnormal risk level, its response logic can be written to generate a honey file to deceive the attacker for a further investigation thanks to the SYSFLOW's *redirect* action. Furthermore, SYSFLOW allows security applications to define their own logic as a new action via the *redirect* action.

For the third challenge, we design SYSFLOW to provide security applications with flexible, global visibility with contexts for risk awareness. To determine the contexts useful for security applications, we collect several context attributes of system objects that could be used to identify malicious behaviors. SYSFLOW leverages the components, i.e., *PCC: Process Context Collector*, *FCC: File Context collector*, *PFM: Process File Mapper*, and *UCC: User Context Collector*, to correlate to contexts. SYSFLOW also utilizes PFM to track and store the relations between them. PFM correlates the contexts of processes and files, and stores for consistent tracking. Such mapping information is not always available in existing monitoring systems because suspicious programs mostly access sensitive resources by invoking another process, which makes it hard to trace the original process. UCC extracts the user context (i.e., usernames) by monitoring specific system calls during login requests. However, the limitation of such visibility service is that every time a system event is matched, a report message should be sent to the controller. To provide the visibility service in such a reactive way will cause significant performance overhead. Thus, instead of solely relying on the controller to handle the context information, SYSFLOW enables a security application to preprocess the context information in SDP with external security functions (e.g., extract the statistics information from a sequence of system events instead of sending every context to the security application).

**Global, Cross-host Visibility.** SC envisions system contexts (collected from SDP) to develop security apps among multiple host systems in the infrastructure by design just as SIEMs do with log collection. A security app can collect contexts by registering flow report handler functions in SC to process flow status reports from installed monitoring flow rules in a target system. In particular, SYSFLOW supports two types of (cross-host) context sharing, i.e., *reactive controller updating* and *proactive packet tagging*. In a reactive updating manner, a security app installs multiple monitoring flow rules in various systems, registers flow reports to acquire system contexts from different host systems, and optionally updates flow rules based on monitored contexts. In a proactive packet tagging manner, a security app can proactively install flow rules to leverage *encode* and *decode* security actions to encapsulate system-level contexts to the tags of outgoing packets and enforce different security actions based on the tags.

## VI. PERFORMANCE OPTIMIZATIONS

Without efficient monitoring and managing system events of interest defined as a flow, SYSFLOW may not be suitable for the Zero Trust model due to the performance overhead. Also, the overhead from the interaction between SDP and SC begets another hurdle. In this section, we introduce how SYSFLOW minimizes the overhead to address the challenge (**C4**).

### A. Efficient Flow Rule Management

The low-level security intents of SYSFLOW represented with a list of system flow rules are embedded in the SYSFLOW flow table. To support both *exact match* and *wildcard match*, a naive solution for the SYSFLOW flow table is to use a bitwise classification through a bitwise comparison between incoming system events and all system flow rules. However, the time complexity of flow table lookups and updates is $O(R)$, such that $R$ is the number of system flow rules installed in the table. As a result, flow rule installation will incur a high latency in the data plane as $R$ increases.

To design an efficient flow table update and query, we adopt the Tuple Space Search (TSS) classification algorithm [46]. The key insight of TSS classification is to realize a flow table as a set of hash tables. In each hash table, it stores the hashed key for each specific system flow rule with the same mask. Suppose all the flows in a SYSFLOW flow table matched on the same fields in the same way, e.g., all flows match the source and destination system object but no other fields. In such a case, TSS implements a flow table as a single hash table. If a new flow with a different match is added, TSS generates another hash table that handles the new match for the flow. To denote matched fields, we use a 3-bit mask to specify the range of each hash table. Based on the TSS algorithm, Flow Table Manager can provide an efficient flow table management, e.g., flow rule update/lookup, with time complexity of $O(1)$. [1] As shown in Section IX-A2, the maximum number of flow rules to cover all file operations from the fresh installation of Linux is less than 100,000 and we evaluate the efficiency of our flow rule management in the evaluation.

### B. SYSFLOW *External Security Function*

The insight of SYSFLOW external security functions is to offload the system event processing functions/code of a security app into SDP to reduce the interactions between SC and SDP. For example, based on *deny by default*, Zero Trust applications may request missed flows to be sent to SC for the further decision. If we report each of *flow-miss* events to SC for analysis, the round-trip latency will greatly delay the attack investigation and response. To address this, we design SYSFLOW external security functions that can run any security function code written by the SYSFLOW applications in a safe sandbox (eBPF VM) in the data plane instead of the controller. Also, SYSFLOW external security functions are designed to be securely and dynamically installed in SDP. By leveraging SYSFLOW external security functions, not only SYSFLOW can move security logic from a security application to SDP but also can support processing fine-grained context information in SYSFLOW external security functions on behalf of the security applications. We showcase how a reactive security application can be optimized to enhance performance through SYSFLOW external security functions in Section IX-B.

## VII. SYSFLOW IMPLEMENTATION

A prototype of SC has been implemented in Java[2] with 4,267 lines of code (LoC) by using Non-Blocking IO (NIO) APIs to achieve high event processing throughput. Currently, SYSFLOW security apps can be developed in Java[3] and instantiated as a module of SC.

We have implemented a prototype of SDP on top of Linux in C with 7,094 lines of code (LoC). SYSFLOW captures a list of system events based on the Linux Security Modules (LSM) framework [51]. LSM hooks are used to generate system events in the SYSFLOW kernel plugin module. It currently supports a set of file, inode, memory, and socket operations. In addition, to support system events in containers, SYSFLOW checks the namespace of processes to pinpoint the container that generates the system events. Note that SYSFLOW is extensible to support more system level events for different operating systems. We will discuss the extensibility of SYSFLOW in more detail in Section X.

**Built-in User Context Support.** UCC enables visibility at the application-level. In SYSFLOW, users can utilize system flow rules and external functions to program their approaches to extract customized user context from the data plane. To reduce the work of users, SYSFLOW provides a general approach to retrieve regular context with minimal effort. Since most context information visible in system calls are passed as parameters, we use a six elements tuple, `<process_name, event_name, parameter_index, context_index, context_length, condition>`, to represent how we may extract the context information. `process_name` represents the name of process. `event_name` and `parameter_index` correspond to the system event and the parameter that will contain the context information. The context information can be located in the parameter with `context_index` and `context_length`. `condition` shows the requirement of system events that should be handled.

For each tuple, SYSFLOW will generate a flow rule to redirect the specific system event to the external function. The external function will check the condition and extract the user context. In our current implementation, we only support the length condition but it is trivial to provide support for complicated expressions or code snippet written in C as the condition is a part of the generated external function. Then, another system event will be generated, which will forward the user context to the controller in a flow report message. For example, the tuple used in UC#3 in Section VIII is `<'nginx', 'io_getevents', 4, USERNAME_INDEX, 0, ">1024 && <1596">`. The tuple will be converted to a system flow rule and an external function. The system flow rule matches all `io_getevents` syscalls and the external function will extract the username and generate another system event with user-context information.

SYSFLOW **Control Messages.** For communications between SC and SDP, we define three types of control messages (shown in Table II), i.e., symmetric messages, control-plane-to-data-plane (CP-to-DP) messages, and data-plane-to-control-plane (DP-to-CP) messages. The control messages are exchanged via a secure channel with SSL/TLS. The details are briefly described in the table.

**Flow Rule Management.** Flow rules are managed in SDP and SC separately. SC can pro-actively and re-actively install or update flow rules in the flow table by using *flow rule modification* messages. A new flow rule is installed in the flow table upon *flow miss* after the lookup of a hashed key and is updated upon *flow hit*. The flow rules are stored in the kernel memory for fast lookup. In the controller side, the management and housekeeping of flow rules freely depend on how SC apps handle them, as most SDN controllers follow. The normal operations of security apps are to store flow rules in the internal storage or database when installing the rules and to update the entries when updating them via the *flow*

[1]The flow rule lookup with TSS needs $T$ hashed memory accesses, where $T$ is a constant value (i.e., 8 in the paper) of the number of tuples. The flow rule update with TSS needs 1 hashed memory accesses.

[2]We note the design of the controller is programming language agnostic.

[3]SYSFLOW currently supports Java-based security apps due to the user-friendliness and generality of Java.

TABLE II
SYSFLOW CONTROL MESSAGES EXAMPLE (CP: CONTROL PLANE, DP: DATA PLANE)

| Control Message | Type | Description |
|---|---|---|
| Host info request | CP-to-DP | Request host information |
| Host info reply | DP-to-CP | Reply host information |
| Flow rule modification | CP-to-DP | Insert/remove/update system flow rules |
| Flow rule status request | CP-to-DP | Request the status of system flow rules |
| Flow rule status report | DP-to-CP | Report the status of system flow rules |
| Ext. Func. modification | CP-to-DP | Add/delete/update SYSFLOW external security functions |
| Ext. Func. report | DP-to-CP | Report the status of SYSFLOW external security functions |

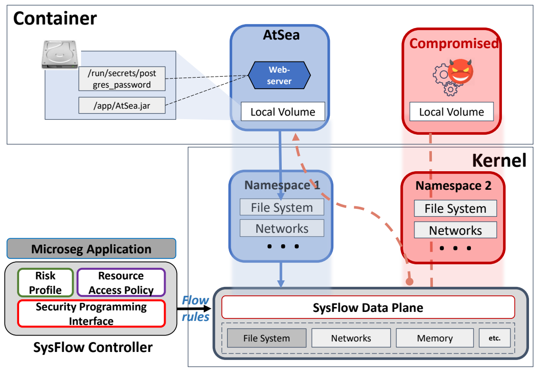

Fig. 4. Micro-segmentation for microservices.

*rule modification* message. Also, SC retrieves the status of the flow table and flows, and the list of flow rules from the flow tables in SDP using *flow rule status request*.

## VIII. SECURITY APP EXAMPLES IN THE REAL WORLD

We demonstrate the usefulness of SYSFLOW in practice. The key innovations of SYSFLOW are three-fold. We enhance the existing system security in the context of Zero Trust by providing admins with easy programmability for system resource access control and risk validation in both generic and container-based systems. Also, SYSFLOW can extend security functionalities flexibly, not limited to the existing capabilities. In addition, SYSFLOW can easily integrate with the existing Zero Trust security through global visibility. We have developed several security apps for real-world security problems based on SYSFLOW. Table III categorizes each app by the capabilities of SYSFLOW. To address each aforementioned innovation, we demonstrate three example apps for container security and three apps for generic system security to show how micro-segmentation and risk awareness with visibility can be implemented using our APIs.

TABLE III
EXAMPLE SYSFLOW APPS. (NOTATIONS: UP (UNIFIED PROGRAMMABILITY), DR (DYNAMIC RECONFIGURATION), FV (FLEXIBLE VISIBILITY), GV (GLOBAL VISIBILITY), PC (PROCESS CONTEXT), UC (USER CONTEXT))

| Name (Type) | Description | Capabilities |
|---|---|---|
| RS (Access Control) | Risk-aware micro-segmentation for containers | UP, PC |
| FCAC (Access Control) | User-context-aware micro-segmentation | UP, PC, UC |
| File Reflector (Cyber Deception) | Reflect suspicious file operations to honeypots | UP, FV |
| CLDLP (Info Flow Control) | System/network information flow based DLP | UP; GV; DR |
| VP (Virtual Patching) | Virtual patching for container-based system | UP, GV, DR |
| CFDAC (Access Control) | Cross-host, context-aware access control | UP, GV, DR |

**UC#1: Risk-aware Micro-segmentation for Microservices.** We first illustrate the SYSFLOW security application that leverages our micro-segmentation and risk profile APIs for the motivating example (Figure 1). As we analyze it through the paper, the system security perimeter provided by the *namespace* isolation is broken when a privileged option is enabled to grant access to system (and other containers) resources by exploiting container vulnerabilities. For example, in an open-source eShop application, AtSea, *app_server* is the container that provides web service to users. If the *app_server* container is compromised, attackers can access "/run/secrets/postgres_password/", which stores the password of the database. With the database password, attackers can access the database and dump the entire database.

```
### Risk-aware Micro-segmentation for Micro-services
...
    Host APP_SERVER_HOST = "10.0.0.1";
    Container APP_SERVER_CONTAINER;//container id of
         app_server

    Void init(){
        ms_create("atsea");
        Match flow_match = [src: *, dst: *, op_type: *];
        ms_alloc("atsea", flow);
        profile = context_profile(Microseg.class.
             getMethod("policy", parameterTypes));
        ms_acl("atsea", profile);
        mc_deploy("atsea", APP_SERVER_HOST);
    }

    Boolean policy(sf_obj src, sf_obj dst, sf_type
         op_type){
        container_id = read("/proc/" + src + "/cgroup");
        if(container_id == APP_SERVER_CONTAINER)
            return true;
        else
            return false;
    }
```

Listing 1. Micro-segmentation example.

To mitigate such threat, we developed a security app to confine file accesses within each container scope and prohibit the malicious requests from the compromised container to *AtSea* containers as depicted in Figure 4. As shown in Listing 1, the application initiates a micro-segmentation for *AtSea* with *ms_create* and includes all system flows targeting directories mapped to the *AtSea* volumes with *ms_alloc*. In addition, the application also creates a context-based risk profile to check if the incoming file access is from a process inside *AtSea* containers (based on namespace information). Finally, the application links the context risk profile to the micro-segmentation with *ms_acl*. The application will automatically translate the micro-segmentation policies to flow rules, which forward all requests inside the micro-segmentation and reject all requests across micro-segmentation. This use case showcases how SYSFLOW users can easily write a ZT app with high-level APIs. Our evaluation shows that the installation of the flow rules in the example took 7.41ms and the average latency of file operations inside the micro-segmentation was increased by less than 5%.

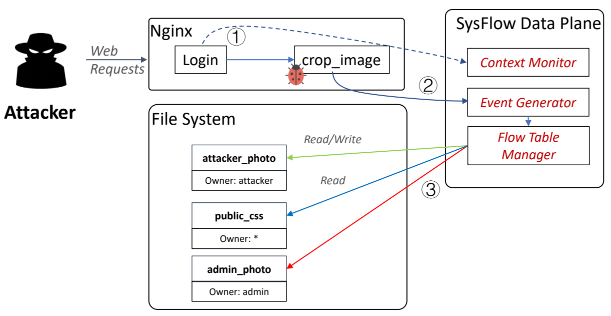

Fig. 5. Workflow of FCAC in SYSFLOW dataplane.

**UC#2: Fine-grained and Context-aware Access Control in Web Applications.** Access control is an essential security approach to protect resources in infrastructure. However, existing security systems mostly enforce access control policies of each application individually with a coarse- or medium-grained user-based authorization scheme due to the lack of visibility and high-level programmability. Also, they are either application-level oriented or system-level oriented. Figure 5 depicts this case. A malicious user can access the private data of other users by exploiting vulnerabilities inside the web application. The threat is from an application-level user to system-level resources. When authentication mechanisms in the web application are broken/bypassed, system-level security mechanisms like role-based access control are only aware of the web application, not the application-level user. Hence, they cannot detect illegal access. We implemented a Fine-grained and Context-aware Access Control (FCAC) application to provide fine-grained access control with the awareness of contexts in different levels. FCAC app in SYSFLOW will install flow rules to restrict accesses to files based on the user profile contexts in the web application. In detail, there are three rules: 1) FCAC authorizes the access if the file belongs to the user; 2) any access to files that do not belong to the user will be denied; 3) Write requests from normal users to public files will be denied. To determine the owner of a file, FCAC monitors file operations to record the creator of the file. The creator is assumed to be the only user of the file. Public files should be created either by a specific user or before FCAC is installed. In our test scenario, we ran WordPress 5.0.3 (CVE-2019-8943 unpatched) with Nginx 1.18.0. To extract the user context (i.e., usernames), FCAC utilizes several system flow rules by monitoring system call parameters during login process. The username will be extracted as a user identifier. We tried to log in the attacker's account and exploit the vulnerability to access a photo uploaded by another user. The access was rejected by SYSFLOW and Nginx returned `404-File Not Found` error to the attacker. The results showcase the effectiveness of cross-application access control policies enforced by FCAC applications in SYSFLOW. We also tested the additional latency introduced by SYSFLOW when 100 users accessing the web application at the same time. On average, FCAC only increases the response time by 4.17%. Listing 2 shows an abstracted example of SYSFLOW applications for FCAC.

```
### Fine-grained and Context-aware Access Control in Web App
...
Host WEB_SERVER_HOST = "10.0.0.1";

monitor_action = String(
    void monitor_action (char* input, int type){
        char username[256];
        char session[256];
        locate_username(input, username);
        locate_session(input, session);
        bind_in_map(username, session);
        bind_in_map(session, pid);
    }
);

access_action = String(
    void access_action(char* file_path, int type){
        char username[256];
        find_in_map(pid, username);
        if(type == file_read){
            if(!is_owner(username, file_path))
                return false;
            else return true;
        } else if (type == file_write) {
            add_owner(username, file_path)
        }
    }
);

void fcac(){
    match_monitor = [src:ANY_SOCKET, dst:*, type:socket_open];
    action_monitor = [compile_action(monitor_action)];
    match_file_access = [src:NGINX, dst:WEB_DIR, type:ANY];
    action_file_access = [compile_action(access_action)];

    installSysFlow(WEB_SERVER_HOST, match_monitor,
                   action_monitor, DEFAULT);
    installSysFlow(WEB_SERVER_HOST, match_file_access,
                   action_file_access, DEFAULT);
}
...
```

Listing 2. An abstracted example of FCAC.

**UC#3: File Reflector with Flexible Visibility.** Cyber deception can be a promising technique to divert attackers away from enterprise/cloud sensitive data for better resource protection and further attack investigation/forensics. Upon SYSFLOW, we implement a security application, namely *file reflector*, which can automate the deployment of decoy resources in the system to lure potential attacks that aim to steal or modify sensitive data (e.g., tax/payroll/password) from protected resources. This example showcases how SYSFLOW is applicable and extensible to more complicated and advanced security problems of ZTA that require continuous validation of access to establish *trust* reliably. Figure 6 demonstrates a high-level idea of how *file reflector* can be realized in SYSFLOW.

```
### File Reflector application
File protected_file = "/etc/passwd"
File decoy_file = "/tmp/decoy/etc/passwd"
target_host = "10.0.0.1"

void file_reflector(){
    match = {src:ANY, dst:protected_file, type:file_open};
    action = [redirect(decoy_file) | report];

    #install system flow rule in target host
    installSysFlow(target_host, match, action, DEFAULT);
}
```

Listing 3. An abstracted example of FileReflector.

By using *redirect* action provided by SYSFLOW as demonstrated in Listing 3, access attempts (i.e., *file_open* system events) to the sensitive file (e.g.,"/etc/passwd") can be added as *multiple* actions to redirect to decoy documents in a programmable fashion. It enables admins flexibly to either force the access redirection immediately or analyze activities

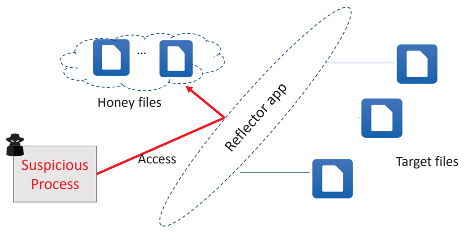

Fig. 6. High-level idea of file reflector app in SYSFLOW.

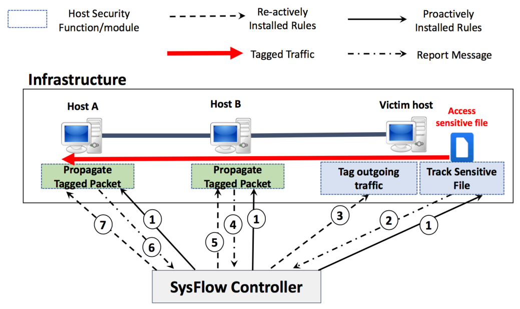

Fig. 7. Infrastructure-wide data leakage investigation realized via SYSFLOW.

on honey files without blocking/confusing normal resource actions. Based on the deception provided by the SYSFLOW app, administrators can conduct more analytic/defensive logic, e.g., through a security application via report, *log*, and an external security function via *external* in a flexible, programmable manner without depending on the raw system logs. They may inspect the interactive system events between the suspicious process that was identified by SYSFLOW Risk Profile and the decoy document to capture follow-up malicious behaviors. We have also evaluated the performance overhead for the deception defense, i.e., the latency to redirect file open attempts from a sensitive file to a decoy one. We observed the average latency of the deception defense is 2.0 microseconds, which is negligible considering the average latency of normal file open accesses (2.5 microseconds). In this case, it can also somehow hide our *redirect* security primitives from an adversary side.

**UC#4: Cross-Layer Data leakage Prevention (CLDLP).** Many advanced attacks may exhibit multi-stage manner across various hosts in a modern computing infrastructure for stealthy and elusive purpose. Unfortunately, it is not an easy integration for the existing fragmented network/system-only security tools for ZTA. In Figure 7 as reported by TrapX [13], suppose that $Host_B$ is *less* protected by firewalls and other security tools, for example, a picture and archive communications system (PACS) designed to easily move medical imagery (e.g., X-rays) throughout the hospital and offices outside. The attacker first compromises $Host_A$ and then $Host_B$ that has less protection, and ultimately infects $Host_C$ (Victim Host) to exfiltrate sensitive files to outside via $Host_B$ (as a stepping stone). Likewise, this multi-stage data leakage scenario is in line with cross-host attacks like GitPwnd [7] when $Host_B$ is a Git server as a stepping stone. In the use case, we showcase how SYSFLOW can help mitigate data exfiltration scenario. To secure sensitive data from exfiltration, information flow tracking techniques are widely used at an infrastructure scale. With the abstraction of system flows, we show a hybrid SYSFLOW app (as illustrated in Figure 7) that enforces cross-layer data leakage prevention. The security app installs system flow rules into the victim host and other hosts in infrastructure (①). For the victim host, SC installs a system flow rule to track the system-level information flow from the sensitive file to any processes. When a sensitive file in the victim host are accessed by other hosts, a report is sent to SC (②). When the controller receives the *report* message from the victim host, it will reactively install an information flow tracking flow rule to encode a tag for all outgoing traffic for the sensitive file.Since the SC has already installed flow rules on other hosts to instruct SDP to report any observation of tagged packets received from the socket (④ and ⑥), when SC receives the *report* from SDP, it responsively installs flow rules to propagate the tag for outgoing traffic (⑤ and ⑦). When attackers in $Host_A$ (e.g., inside a hospital) attempt to retrieve the data in $Host_B$, or when attackers from outside (e.g., offices outside the hospital or the Internet) try to retrieve the data via $Host_B$ (as a stepping stone), SC will be notified that the sensitive file is about to leak from $Host_B$. Besides, the non-stepping stone host, $Host_A$, is also applied with the data leakage prevention rules easily by this infrastructure-wide mechanism. Listing 4 shows an abstracted example of SYSFLOW applications for CLDLP, which impedes data exfiltration of a sensitive file (i.e., "*/usr/data/medical_info*") and Section IX-B elaborates how we optimize performance by replacing the reactive programming with proactive programming based on external functions.

```
### Cross-Layer Data Leakage Prevention app
File sensitive_file = "/usr/data/medical_info"
Tag tag = "dlp";
match1 = {src:ANY, dst:sensitive_file, type:file_open}
match2 = {src:ANY, dst:ANY_SOCKET, type:socket_read}
match3 = {src:ANY, dst:ANY_SOCKET, type:socket_write}
# action definitions for report, encode, decode, net(info) flow
...
netflow_match = {src:ANY, dst:internet, TOS: tag}

void information_flow_tracking(){
    #install a system flow rule to monitor processes
    #trying to the sensitive file
    installSysFlow(host1, match1, report_action,
        DEFAULT)

    #install a system flow rule to report any process
    #read from tagged socket
    installSysFlow(host2, match2, report_action,
        DEFAULT)
    installSysFlow(host3, match2, report_action,
        DEFAULT)

    #install a network flow rule in gateway switch to
    #block outgoing traffic towards Internet with tag
    installNetFlow(Gateway, netflow_match, [deny])
}

#register handler to encode tags for outgoing traffic
callback handleSysFlowReport(report){
    match = report.match
    host =  report.host
    if match == match1 or match == match2 then
        #install system flow rule to tag any process
        #reads the sensitive file or read tagged socket
        installSysFlow(host, match3, encode_action,
            DEFAULT)
}
```

Listing 4. An abstracted example of CLDLP.

**UC#5: Virtual Patching.** Virtual patching (VP) [21] is a security policy enforcement layer which prevents the exploitation of a known vulnerability in a timely manner without waiting for an actual patch release. It analyzes suspicious activities and intercepts attacks in transit so that malicious access/traffic cannot reach the victim's resources. The deployment of the vulnerable container, however, is *dynamically* decided at run-time by container orchestration tools, e.g., Docker Swarm [5] or Kubernetes [10], which makes the existing security tools difficult to realize this technique. In this use case, we show-case how SYSFLOW, thanks to *Dynamic Reconfiguration* in particular, easily implements a virtual patch. Suppose that the vulnerability of a web server, *nginx*, in Figure 4 is reported at CVE, admins need a prompt virtual patch for a server without modifying *nginx*. To solve this, SYSFLOW can track *nginx* using *Dynamic Reconfiguration* and a virtual patching application can be written with a defense code to block the signature payload of the vulnerability. To this end, VP first installs a flow rule to identify whether the vulnerable *nginx* container is running in the swarm cluster of the host. Upon detecting a process related to vulnerable *nginx*, VP dynamically installs virtual patching rules to block an attack payload containing the discovered path traversal vulnerability in the corresponding host systems. This procedure can be done quickly by admins/developers on top of SYSFLOW without disrupting the current operation of *nginx*, which emphasizes the benefits of SYSFLOW's programmability, holistic visibility, and dynamic reconfiguration. Listing 5 shows an abstracted example of SYSFLOW applications for Virtual Patching.

```
### Virtual patching example (for host1,host2,host3)
vulnerable_process = "nginx"
legal_path = "/var/www/*"
...

function vulnerability_monitor{
    match = {src:vulnerable_process, dst:ANY_FILE,
        type:ANY}
    action = [report]
    # install flow rules for host1, host2, host3
    installSysFlowRule(host1, match, action, DEFAULT)
    ...
}

callback virtual_patch_nginx_path_traversal(report){
    target_host = report.host
    match = {src:vulnerable_process, dst:ANY_FILE,
        type:file_open}
    action = [deny]
    installSysFlowRule(target_host, match, action,
        DEFAULT)

    match = {src:vulnerable_process, dst:legal_path,
        type:file_open}
    action = [allow]
    installSysFlowRule(target_host, match, action,
        DEFAULT)
}
```

Listing 5. An abstracted example of Virtual Patching.

**UC#6: Cross-host, Fine-grained, and Dynamic Access Control (CFDAC).** Access control is an essential security approach to protect important resources in infrastructure. However, existing security systems mostly enforce access control policies in a single host with a coarse- or medium-grained authorization scheme, which may break down in the context of dynamic environments or complex access policies. Instead, SYSFLOW can enable fine-grained access control applications with the awareness of cross-host contexts in a dynamic manner. Such contexts may include host identity, user identity, time, file visit history, file name, and so on as well as Zero Trust contexts (e.g., authentication, permission, etc.). We assess such a capability by defining dynamic access control policies enforced by a Cross-host, Fine-grained, and Dynamic Access Control (CFDAC) application. When a running process attempts to access the DB service remotely, the CFDAC app in SYSFLOW installs corresponding flow rules to restrict accesses based on the device and time contexts. First, the CFDAC authorizes the access if it is within work hours (e.g., from 9:00 am to 5:00 pm) and only loads trust libraries (e.g., in a whitelist). Second, any access request from processes is denied if it is not within work hours. Finally, the access is denied if the process loads uncertainty libraries. Listing 6 shows the simplified example of CFDAC that utilizes time contexts to restrict database access.

```
### Cross-Host Fine-grained Access Control (for server, client)
Process db = "/usr/bin/mysql"
match1 = {src:ANY, dst:ANY_FILE, type:open}
match2 = {src:db, dst:ANY_SOCKET, type:socket_read}
...
# install flow rule to monitor file access
installSysFlowRule(client, match1, report, DEFAULT)
...

callback timer(start_time){ # start at 09:00
    # revoke flow rule in server
    revokeSysFlowRule(server, match2, DEFAULT)

    # install flow rule to allow access at 09:00
    installSysFlowRule(server, match2, allow_act, DEFAULT)
}

callback timer(end_time){  # ends at 17:00
    # revoke flow rule in server
    revokeSysFlowRule(server, match2, DEFAULT)

    # install flow rule to deny access after 17:00
    installSysFlowRule(server, match2, deny_act, DEFAULT)
}
```

Listing 6. An abstracted example of CFDAC.

## IX. EVALUATION

In this section, we conduct evaluations to (i) measure the performance overhead of SYSFLOW to show the minor overhead on normal system operations, (ii) show the efficient handling of flow rule updates, and (iii) verify the scalability of the SYSFLOW controller. In the following evaluations, by default, we hosted the SYSFLOW controller running on Ubuntu 18.04 with 2 cores of Intel i5 9600K CPU and 16 GB RAM and for other hosts, we ran Ubuntu 18.04 with 2 cores of Intel i5 9600K and 8 GB RAM.

### A. Data Plane Performance Measurement

In this section, we present the performance of SDP for micro-benchmark tests, macro-benchmark tests, and scalability tests. In the following evaluations, we leverage *baseline* to refer to systems running an unmodified Linux kernel.

*1) Benchmark Results:* We used LMBench [11] to evaluate the run-time performance of system operations. Table IV depicts the comparison between the baseline and SYSFLOW with the applications in Section VIII deployed. For two of the three primary file operations (i.e., *read*, and *write*), the introduced overhead is reasonably low. Our evaluation results indicate that SYSFLOW mainly impacts the file operations and mmap. But for other operations SYSFLOW only introduces negligible overhead.

TABLE IV
LMBENCH RESULTS FOR SDP.

| System Operation | Baseline | SysFlow | | |
|---|---|---|---|---|
| | | best | avg | worst |
| Latency of system operations in ms (smaller is better) | | | | |
| file read | 0.1913 | 0.1960 | 0.1991 (+4.08%) | 0.2106 |
| file write | 0.1879 | 0.1901 | 0.2036 (+8.36%) | 0.2273 |
| file open/close | 0.4875 | 0.5124 | 0.5482 (+12.45%) | 0.6124 |
| file create (0k) | 2.9717 | 2.9453 | 3.0977 (+1.04%) | 3.3219 |
| file create (10k) | 6.6968 | 6.8053 | 7.6874 (+14.79%) | 7.4085 |
| file delete (0k) | 4.9388 | 4.9744 | 5.2591 (+6.49%) | 5.6933 |
| file delete (10k) | 3.4770 | 3.5102 | 3.5924 (+3.32%) | 3.8210 |
| syscall | 0.0359 | 0.0354 | 0.0361 (+0.56%) | 0.0371 |
| mmap latency | 3418.0 | 3677.0 | 3854.7 (+12.78%) | 4164.5 |
| pipe latency | 2.0840 | 2.1017 | 2.1586 (+3.58%) | 2.3658 |
| Socket throughput pps (larger is better) | | | | |
| socket I/O | 885 | 890 | 874 (-1.24%) | 842 |

TABLE V
MACRO-BENCHMARK RESULTS FOR SDP.

| Type | Baseline | SysFlow | | |
|---|---|---|---|---|
| | | best | avg | worst |
| Web Server (Nginx) Performance | | | | |
| 10K total requests with 500 concurrent connections | | | | |
| Requests per second | 3,533 | 3502 | 3,478 (-1.56%) | 3407 |
| Time per request (ms) | 141 | 143 | 147 (+4.26%) | 154 |
| File Transfer (wget) Performance for 1 GB file | | | | |
| Time to Complete (s) | 21.8 | 21.9 | 22.3 (+2.29%) | 23.3 |
| Throughput (MB/s) | 48.0 | 47.7 | 46.5 (-3.13%) | 45.3 |
| Database (MySQL) Performance with 1M records | | | | |
| Transactions per second | 540.6 | 538.6 | 535.1 (-1.02%) | 529.7 |
| E-shop micro-service | | | | |
| Requests per second | 2472 | 2452 | 2401(-2.87%) | 2397 |
| latency(ms) | 44.54 | 44.87 | 45.17(+1.41%) | 45.62 |

We tested SDP with macro-benchmarks including web server performance, file transmission performance, and database online transaction processing performance. In all of those tests, we run SDP in both server side and client side with 1000 system flow rules with default *ALLOW* actions. For the web server performance, we used a host running ApacheBench [2] to test the performance of a Nginx server by sending 10,000 requests with 500 concurrent connections. To test the file transmission performance, we used wget benchmark [22] in a host to test the transmission of a 1 GB file from a server. For the database performance, we used sysbench [18] to test a database server with 1 million records. In addition, we tested the performance overhead of SYSFLOW upon cloud native e-shop applications with light-4j [14]. We used ab [1] on the same machine to simulate 4093 users and recorded the number of requests per second and latency of the service. The results (Table V) show SDP introduces negligible overhead on the operations of real-world applications even across different hosts.

*2) Scalability with Flow Rules:* We tested the scalability of SDP using different numbers of system flow rules. We increased the number of system flow rules from 1,000 up to 100,000 in our experiments (the maximum number of flow rules is set to 100,000 based on the fact that the number of flow rules to hit every default file operation can be set less than 100,000). Since most security practice can be realized by flow rules with specific actions to specific system objects, the actions of these rules are all *allow* and most of them will only match specific system objects (e.g. specific file). The results show that the number of system flow rules does not make significant differences to the performance. Besides the static flow rules, we also tested SYSFLOW with global file access control, in which we can get more practical and dynamic rules as the previous benchmark results show that the file operations impose the majority of overhead. We deployed an application to approve file access based on user group, as defined in Linux. The application has no predefined flow rule. Instead, all flow rules will be generated at runtime. On a fresh installation of Ubuntu 18.04 with 332,358 files, SYSFLOW installed 74,519 flow rules (designed to look up all flow entries for each file with a reduced number of rules using *wildcard*) in 37.20s after the system was booted. After that, the delay of file operations introduced by SYSFLOW is reduced to 4.71% on average. The results show that SYSFLOW has no significant overhead when most flow rules have been installed. In addition, we tested the memory overhead introduced by SDP through the *top* Linux command with different numbers of system flow rules as the memory operation is another major performance overhead from the previous evaluation. The result shows the memory usage is about 400 KB for 1,000 flow rules and it grows linearly with the number of system flow rules inserted. Hence, SDP is scalable to contain system flow rules for various system security intents.

### B. Efficiency of Flow Rule Update

**Reactive App vs. Proactive App.** SYSFLOW provides external security functions to reduce the latency during communications between SDP and the SC. We used the external function to optimize the reactive implementation of the Data Leakage Prevention (illustrated in Figure 7) that installs the flow rules only in response to report messages that contain access information to sensitive files. We optimized this app in a proactive way by offloading the operations of reactive flow installation as a SYSFLOW external security function. For a reactive application, the delay time $t_r$ of the application consists of three parts: 1) the communication between the data plane and the controller, which is 3.903ms in our evaluation, 2) the processing time in the controller for handling a report and sending a new flow rule to the data plane, which is 0.007ms, and 3) the processing time in the data plane for installing the new flow rule, which is 2.447ms. In contrast, for a proactive application, the delay time $t_p$ of the application comprises only two parts: 1) the processing time that the external function handles a report and sends a new flow rule back to the flow table manager, which is 0.004ms and 2) the time that the flow table manager installs the new flow rule, which is 2.091ms. We implemented both approaches. The result shows that the latency is reduced by about 60%, which is approximately the first part of delay time of the reactive implementation. We found that $t_{p1}$ is smaller than $t_{r2}$. The reason is that external security functions are implemented in C, thereby slightly faster than applications implemented in Java.

**Dynamic Policy Programming.** The comparison of reactive and proactive apps shows that the dynamic policy programming of SYSFLOW can significantly reduce the time cost for dynamic reconfiguration. We measured the overhead introduced by the dynamic reconfiguration when container instantiation occurs. We instantiated the identical Ubuntu 18.04 image with Docker on two hosts but only one runs with SDP. In the evaluation, SC installs one flow rule for each container to deny any access to file `/etc/passwd` using a proactive app. Then, we measured the container starting time on the two hosts by instantiating the image 100 times. For the host without SDP, the average starting time is 0.374s. For the host with SDP, the average starting time is 0.381s. The results show that the overhead introduced is about 1.9%. In practice, the overhead could be larger since more flow rules should be deployed. However, the overhead will not increase significantly since each additional flow rule will only introduce about 2ms latency.

**Sensitivity of Flow Rule Update to Residual Flow Rules.** We added a test code snippet in SDP Daemon that leverages the *gettimeofday* API with a microsecond timestamp to measure the latency of inserting/updating/deleting 10,000 flow rules cumulatively (from 10,000 flow rules to 50,000 flow rules defined as discussed in Section IX-A2). We repeated the measurement five times for each case. Figure 8 shows the average latencies for Flow Table Manager to handle *flow rule modification* messages. We can observe that Flow Table Manager can efficiently handle all types of *flow rule modification* messages, e.g., the average latency is around 1.7 ms for inserting 10,000 flow rules when 10,000 flow rules have been inserted. Thus, the time of inserting rules is almost negligible. We also observed that the residual flow rules do not have negative effects on the flow rule modifications since the latencies for different scenarios are in a constant trend. As a conclusion, the design and implementation of the Flow Table Manager can support a rapid reconfiguration for high-level security intents.

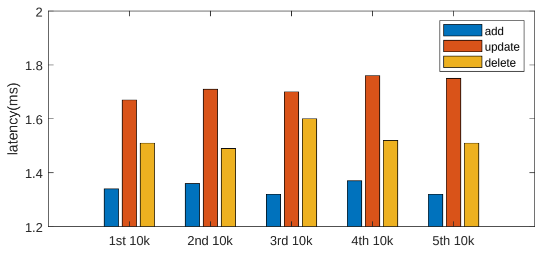

Fig. 8. Composition of system flow rules.

## C. Controller Scalability

We tested the event processing throughput of a single SC. The SC ran a stateful app that installs a new flow-mod message to allow the read operation upon receiving a flow report. For the SDP, we built 3 hosts as event generators. Our event generators send flow rule status report messages to the controller as fast as possible, which tried to read every file in the file system. We measured the observed memory and time cost of each flow mod message. In the evaluation, we installed 223,496 flow rules in total to the 3 hosts. The average memory cost for each flow mod message is 479 KB. It is worth noting that the memory consumed by the flow mod message will be released when the message has been sent to the host. Also, the memory cost is independent between different flow mod messages, which means the needed memory is increasing linearly as the number of generated events per second is increasing. For the time cost, most flow mod messages in our evaluation are sent within 0.01 ms. The time cost of handling the flow report is determined by the complexity of the application. Thus, we also measured the time cost in complicated applications as shown in Section VIII and the flow mod messages are all sent within 0.4 ms after the flow report is received. The results show that the memory cost of handling each incoming flow report and generating a flow mod message is about 480 KB and the time cost is at most 0.4 ms.

According to the report from Solarwinds [6], a host system running SDP generates 5 events per second on average, which means the average number of flow reports from each host system will be 5 in most cases. Hence, 1 GB RAM can support the memory needed for around 5,000 host systems running SDP with the applications. Since SYSFLOW is mainly limited by the RAM of the controller, which is the cheapest part for servers nowadays, SC can be easily scalable in most scenarios.

## X. Discussion

**Diverse Operating System Support.** Despite the Linux-based implementation, we consider the design of SDP is general. This generality comes from our abstraction of system events from low-level system activities (i.e., system calls). Our abstraction bridges the semantic gap between different types and versions of operating systems to make SYSFLOW a uniform framework for different operating systems. The clear separation between the low-level system activities and the security directives, in addition, makes SYSFLOW easily portable to other operating systems as long as corresponding plugins are implemented. One meaningful future work is to extend SYSFLOW to support more operating systems, e.g, Windows and Mac.

**Application-level Event Control.** Currently, SYSFLOW monitors and controls the interactions between processes and system resources (in the kernel). However, we consider the flow-level control scheme proposed by SYSFLOW is general, which can be extended to process and control application-level events (e.g., library calls).

**System Circumvention.** The SYSFLOW Controller could be a primary target for attackers since it is a central point of failure. A basic countermeasure is to enforce authentication, authorization, and access control to prevent unauthorized intrusive activities to the controller. One may use role-based access policies that are audited and reviewed consistently and any modification to them must be audited regularly. Also, the state-of-the-art security practices could be conducted to harden SC as part of ZTA control plane components. In addition, the attacker may attempt to get a root privilege to disable or bypass SYSFLOW external security functions installed in a host system. However, SYSFLOW allows only trusted, vetted code from SYSFLOW Controller to run as external functions in eBPF VM, which prevents unprivileged users/programs from modifying or running a JIT compiler with untrusted, unvetted code by installing system flow rules, which neutralizes the attack in advance.

**Future ZT Features to Add.** Zero Trust is not a single architecture but a set of ideas designed to minimize uncertainty without the assumption of permanent trust [43]. One feature of ZTA is the continuous authentication of a user, device, applications, etc., which is not the goal of this paper. Also, ZTA may support a variety of methods to continuously evaluate and verify trust, such as statistical analysis, machine learning techniques, which are not the focus of this paper either. Instead, we provide useful programming interfaces to write policies and algorithms. We leave other ZT features as our future work.

**Mutual Authentication** Zero Trust advocates mutual authentication, for example, checking the identity and integrity of devices, access to apps and services based on the confidence of device identity and health in combination with user authentication. However, SYSFLOW includes only the identity management component for user/container/process/host now. The reason is that the focus of our paper is to provide a novel framework for programmable system security to realize Zero Trust instead of implementing and integrating all existing Zero Trust components into the framework. Many mainstream ZTAs have already supported mutual authentication for user/device/app/service, which is complementary to our research and could be used/integrated to work together with SYSFLOW. One future direction includes the integration of our framework with the existing ZTA components to serve system security for Zero Trust.

**Formal Verification.** The desired behavior of SYSFLOW entirely depends on the flow rules and security functions issued by applications running on the controller. This may simplify the formal modeling and verification of SYSFLOW architecture. The verification tools for SYSFLOW architecture as a guiding principle to verify the reference monitor and the whole framework can be built either on flow control references between the control plane and the data plane or on each plane just like many SDN approaches [28], [34]. We leave this as our future work.

## XI. Related Work

To date, the increasing number of Zero Trust Architecture has emerged in the industry, such as Google Beyond Corp [8], Palo Alto Zero Trust [16], and Zero Trust eXtended [23], which help organizations better serve a more pragmatic, step-by-step approach as incremental deployment. However, the majority of existing ZTAs have mainly focused on network security and there has been little work to implement a Zero Trust framework for system security like SYSFLOW. Similar to network-based approaches, it does not necessarily mean that there has been no research to implement some ZT principle. Many prior works have touched some relevant elements that could be applied to ZTA for system security.

**Information Flow Control Systems.** Some previous works, e.g., HiStar [54], Asbestos [27], and Weir [40], propose to enforce Decentralized Information Flow Control (DIFC) at the OS level by using labels to define security/integrity contexts and restrict information flows between kernel objects. These solutions typically have limited low-level programmability for protecting information flows between system objects. They focus on information flow tracking and reasoning but suffer from insufficient capabilities to handle the dynamics, i.e., visibility with contexts. Complementing these works, SYSFLOW can provide flexible programmability based on the system flow model and the user's algorithm/logic.

**System Resource Access Control.** Existing Unix/Linux systems embed security kernel modules (e.g., SELinux [17] and AppArmor [3]) to allow users to write mandatory access control (MAC) policies to protect system resources. To enable fine-grained access control, process-level firewall [49], [50] is proposed to prevent system resource attacks by enforcing more fine-grained access control. However, compared with SYSFLOW, these approaches have limited local view and lack the capabilities of dynamic reconfiguration. Some recent works [37], [39] propose approaches to overcome limited visibility inside containers. However, deploying current MAC approaches to container-running systems is still a burden of manual and static labeling and configurations [25] in coping with the security over the dynamics of microservices without a programmable framework like SYSFLOW.

**Security Monitoring and Correlation Systems.** Alert correlation is proposed by many existing works [48], [53] and SIEM systems, such as LogRhythm [12] and IBM QRadar [9], which correlate log events collected from various sources by using different indicators of compromise. Also, several studies [30], [38] also target real-time attack story constructions from system-level logs. Other log-based approaches particularly for APT [29], [52] exploit the dependency/causality relationships of system events defined from interactions among system objects (processes, files, network connections) in audit logs to aggregate and reduce the number of log entries while preserving forensic analysis. A recent platform, sysdig [19], captures system events through a small driver leveraging a kernel facility called tracepoints for container monitoring inside a container. Complementing these works, SYSFLOW can program the entire operations without log-related burden and enable a realtime response, which fits to the ZT principle.

## XII. Conclusion

SYSFLOW presents a novel system security development framework for programmable ZT security control of host system activities at runtime. It offers unprecedented and unified programmability for users to achieve their dynamic security needs. The evaluation shows that SYSFLOW is useful to design diverse Zero Trust system security apps and only introduces minor run-time overhead.

## Acknowledgement

This material is based upon work supported in part by NSF under Grant No. 1700544, 2148374, 2226339, and 2129164, and ONR Grant No. N00014-20-1-2734. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and ONR.

## References

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.
[2] ApacheBench. https://httpd.apache.org/docs/2.4/programs/ab.html.
[3] Apparmor linux application security. https://wiki.ubuntu.com/AppArmor.
[4] Cisco zero trust. https://www.cisco.com/c/en/us/products/security/zero-trust.html/.
[5] Docker swarm orchestration tool. https://docs.docker.com/engine/swarm/.

[6] Estimating Log Generation for Security Information Event and Management. http://content.solarwinds.com/creative/pdf/Whitepapers/estimating_log_generation_white_paper.pdf.

[7] Gitpwnd. https://github.com/nccgroup/gitpwnd.

[8] Google beyondcorp. https://cloud.google.com/beyondcorp/.

[9] IBM QRadar SIEM. https://www.ibm.com/us-en/marketplace/ibm-qradar-siem.

[10] Kubernetes, production-grade container orchestration. https://kubernetes.io/.

[11] LMbench. http://lmbench.sourceforge.net/.

[12] LogRhythm. . https://logrhythm.com/.

[13] Medjack to launch stepping-stone data ex-filstration. https://www.computerworld.com/article/2932371.

[14] micro-service benchmark framework. https://github.com/networknt/microservices-framework-benchmark.

[15] OpenFlow specification. . https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[16] Palo alto zero trust. https://www.paloaltonetworks.com/network-security/zero-trust/.

[17] Selinux. https://github.com/SELinuxProject.

[18] sysbench. http://manpages.ubuntu.com/manpages/trusty/man1/sysbench.1.html.

[19] sysdig. https://sysdig.com/.

[20] SysFlow. https://github.com/successlab/sysflow.

[21] Virtual patching best practices. https://owasp.org/www-community/Virtual_Patching_Best_Practices.

[22] WGET BENCH. http://www.project-open.com/en/benchmark-wget.

[23] Zero trust extended (ztx). https://www.forrester.com/report/The+Zero+Trust+eXtended+ZTX+Ecosystem/-/E-RES137210/.

[24] Adam Bates, Dave (Jing) Tian, Kevin R.B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Conference on Security Symposium (SEC)*, Austin, TX, USA, August 2015.

[25] Maxime Belair, Sylvie Laniepce, and Jean-Marc Menaud. Leveraging Kernel Security Mechanisms to Improve Container Security: a Survey. In *International Conference on Availability, Reliability and Security (ARES)*, Canterbury, UK, August 2019.

[26] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[27] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *ACM symposium on Operating systems principles (SOSP)*, Brighton, UK, October 2005.

[28] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM Sigplan Notices*, 46(9):279–291, 2011.

[29] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection. In *USENIX Conference on Security Symposium (SEC)*, Baltimore, MD, USA, August 2018.

[30] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott D. Stoller, and V.N. Venkatakrishnan. SLEUTH: Real-time attack scenario reconstruction from COTS audit data . In *USENIX Conference on Security Symposium (SEC)*, Vancouver, BC, Canada, August 2017.

[31] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking. In *ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, USA, October 2017.

[32] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking. In *USENIX Conference on Security Symposium (SEC)*, Baltimore, MD, USA, August 2018.

[33] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, USA, October 2004.

[34] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: verifying network-wide invariants in real time. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Lombard, IL, USA, April 2013.

[35] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *ACM SIGOPS symposium on Operating systems principles (SOSP)*, Stevenson, WA, USA, October 2007.

[36] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX Annual Technical Conference (ATC)*, Boston, MA, USA, June 2001.

[37] Fotis Loukidis-Andreou, Ioannis Giannakopoulos, Katerina Doka, and Nectarios Koziris. Docker-Sec: A fully automated container security enhancement mechanism. In *International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2018.

[38] Sadegh M. Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V.N. Venkatakrishnan. HOLMES: Real-time APT Detection through Correlation of Suspicious Information Flows . In *IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2019.

[39] Amith Raj MP, Ashok Kumar, Sahithya J Pai, and Ashika Gopal. Enhancing security of docker using linux hardening techniques. In *International Conference on Applied and Theoretical Computing and Communication Technology (ICATCCT)*, Bengaluru,Karnataka,India, July 2016.

[40] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on Android. In *USENIX Conference on Security Symposium (SEC)*, Vancouver, BC, Canada, August 2016.

[41] Weizhong Qiang, Jiawei Yang, Hai Jin, and Xuanhua Shi. Privguard: Protecting sensitive kernel data from privilege escalation attacks. *IEEE Access*, 6:46584–46594, 2018.

[42] Michael Reeves, Dave Jing Tian, Antonio Bianchi, and Z. Berkay Celik. Towards Improving Container Security by Preventing Runtime Escapes. In *IEEE Cybersecurity Development (SecDev)*, October 2021.

[43] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. *NIST Special Publication 800-207*, 2020.

[44] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integirty for Commodity OSes. In *ACM symposium on Operating systems principles (SOSP)*, Stevenson, WA, USA, October 2007.

[45] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, Dave Andersen, and Jay Lepreau. The flask security architecture: System support for diverse security policies. In *USENIX Conference on Security Symposium (SEC)*, Washington, DC, USA, August 1999.

[46] Venkatachary Srinivasan, Subhash Suri, and George Varghese. Packet Classification Using Tuple Space Search. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, Cambridge, MA, USA, August 1999.

[47] Genc Tato, Marin Bertier, Etienne Rivière, and Cédric Tedeschi. The eigentrust algorithm for reputation management in p2p networks. In *International conference on World Wide Web (WWW)*, Budapest Hungary, May 2003.

[48] Fredrik Valeur, Christopher Kruegel Giovanni Vigna, and Richard Kemmerer. Comprehensive approach to intrusion detection alert correlation . *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 1(3):146–169, 2004.

[49] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. Jigsaw: protecting resource access by inferring programmer expectations. In *USENIX conference on Security Symposium (SEC)*, San Diego, CA, USA, August 2014.

[50] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. Process Firewalls: Protecting Processes During Resource Access. In *ACM European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, April 2013.

[51] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Module Framework . In *Ottawa Linux Symposium (OLS)*, Ottawa, Canada, June 2002.

[52] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High Fidelity Data Reduction for Big Data Security Dependency Analyses. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[53] Tingfang Yen, Alina Mihaela Oprea, Kaan Onarlıoglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. Beehive: Large-scale Log Analysis for Detecting Suspicious Activity in Enterprise Networks . In *Annual Computer Security Applications Conference (ACSAC)*, New Orleans, LA, USA, December 2013.

[54] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, USA, November 2006.