Building a Security OS With Software Defined Infrastructure

Guofei Gu
Texas A&M University
guofei@cse.tamu.edu

Hongxin Hu
Clemson University
hongxih@clemson.edu

Eric Keller
University of Colorado at Boulder
eric.keller@colorado.edu

Zhiqiang Lin
University of Texas at Dallas
zhiqiang.lin@utdallas.edu

Donald E. Porter
The University of North Carolina at Chapel Hill
porter@cs.unc.edu

ABSTRACT

The recent emergence of Software-Defined Infrastructure (SDI) offers a number of useful tools for managing, monitoring, containing, shepherding, and recovering computing units within an enterprise, cloud, or data center. As SDI utilities grow and the types of resources that can be abstracted into software-managed control and data planes increase, there is a pressing need for datacenter-level operating systems (OSes). Such a datacenter-level OS can further abstract and easily capture higher-level policy goals, and push them down to different types of hardware and software, ranging from application processes to storage and networking. This paper thus proposes S^2OS, an SDI-defined Security OS, which offers an easy-to-use, programmable security model for monitoring and dynamically securing applications. We anticipate S^2OS could unlock a wide range of unprecedented security opportunities, including fine-grained and dynamic security programmability at infrastructure scale, and information flow tracking across an entire infrastructure.

KEYWORDS

Software-defined infrastructure, security operating system

ACM Reference format:
Guofei Gu, Hongxin Hu, Eric Keller, Zhiqiang Lin, and Donald E. Porter.2017. Building a Security OS With Software Defined Infrastructure. In *Proceedings of APSys '17, Mumbai, India, September 2, 2017,* 8 pages.
DOI: 10.1145/3124680.3124720

1 INTRODUCTION

In recent years, there has been a rapid and dramatic paradigm shift in computing from static control systems, often implemented in hardware, to dynamic, easily-reconfigurable software-defined systems. Examples of this shift include multi-tenant clouds, Software-Defined Networking (SDN) [30], Network Function Virtualization (NFV) [2], and Software-Defined Radios (SDR) [47]. A key enabling technology for this shift is virtualization at many layers, including both the OS and networking. For instance, hardware-centric infrastructure requires installing and managing the physical computers and devices, where each installed computing device cannot be moved around or easily modified. With Software-Defined Infrastructure (SDI) [26], we now have a software-defined control layer (often called the virtualization layer) for managing, monitoring, containing, shepherding, and recovering all computing units in software. As a result of this more flexible computing infrastructure, modern data centers and clouds can make more effective use of the available computing power, storage space, and network bandwidth.

Given the success of SDI in automating and simplifying dynamic management of computing hardware, we believe the next frontier is considering how SDI can change the practice of security administration. Historically, many of critical systems have been developed with security as a reactive add-on, rather than a fundamental design goal. Consequently, the security mechanisms are often fragmented, hard to configure, and hard to verify, which makes it difficult to defend against various cyber attacks.

The current generation of hardware-centric, single-system designs have a number of fundamental security challenges. First, existing security mechanisms are often tightly coupled with legacy systems, making these security mechanisms hard to protect from the rest of the system. As such, often times, they share the same set of privileges, and a security breach of the user applications or the operating system can lead to the breach of the security mechanisms. Second, current defenses often focus narrowly on one attack vector, such as a network intrusion or input validation in the kernel; when behavior is not clearly malicious, it is difficult, if not impossible, to share information and coordinate across these multiple layers of defense. Finally, a number of practical considerations lead toward homogeneity, which is also often a root cause of automated cyber attacks—once an exploit works for one system, it is likely to work with others. Diversity would mitigate this problem, but only if administration challenges could be addressed.

This paper argues that SDI provides an opportunity to address these issues, rethinking security policies and abstractions at enterprise scale, unconstrained by the limitations of underlying hardware or legacy system software. In addition to the management capabilities of SDI, we also now have most, if not all, of the tools to implement and replicate any application-facing abstractions in software (and often in hardware too).

The Vision of S^2OS. In this paper, we propose S^2OS, an **S**DI-defined **S**ecurity **OS**, which aims to abstract security capabilities and primitives at various layers, thus providing unified programmability

APSys '17, Mumbai, India

978-1-4503-5197-3/17/09…$15.00
DOI: 10.1145/3124680.3124720

and controllability so that any security policy or procedure can be easily and dynamically programmed at infrastructure scale. S^2OS is designed with the following security capabilities in mind:

- *Strong Isolation*. With virtualization of both the OS itself and networking, security mechanisms can be deployed at a new, universal control layer. Within the host, since the guest OS runs on a separate level above the hypervisor, there is a world switch whenever control passes between the two. The hypervisor thus provides strong isolation between the security mechanism and the attacks present in the guest OS. Similarly, the network control plane is separated from the data plane.
- *Full Automation*. When an intrusion is detected, current practice is often to have a human administrator in the loop, or may rely on mechanisms in the system that are vulnerable to subsequent attacks. By moving security mechanisms into the virtualization layer, we can program automated counter-measures in advance, which execute at a plane below the attacked, external-facing software. More importantly, we can automate and coordinate both host-level and network-level security mechanisms to maximize the coverage and accuracy of defense.
- *Complete Visibility*. The S^2OS control plane is designed to give full visibility into the entire infrastructure, including the memory, networking, and file system state of each running application in the system.
- *High Flexibility*. In S^2OS, virtual hardware in hosts or networks can be quickly altered, reconfigured, replicated, and rolled back for any security purposes. At the application layer of S^2OS, infrastructure managers can easily and flexibly program fine-grained, dynamic security control policies.
- *Trustworthiness*. There is a lack of mutual attestation in existing platforms. S^2OS exports trust measurement as a first-class system primitive between software and the underlying hosting platforms, and provide mutual attestation between a virtual platform and application software.
- *Maneuvering and Diversity*. With a layer below the OS and a control over various networking and software stacks, we can now maneuver a computing unit across the entire infrastructure using a moving target defense. We can also offer diversified environments from the lower layer to the networking, OS, and applications.

Similar to how traditional operating systems manage various hardware resources for user applications, S^2OS abstracts the control layer of SDI and provides APIs for programmable security across an entire infrastructure. More specifically, S^2OS will be the first unified framework to provide security control over various enterprise, cloud, data-center computing and network units, including fine-grained application process executions (e.g., through control agents at hypervisor, container, or library OS), and application-aware network flows (e.g., through control agents at extended Open vSwitch software).

The “killer application” for S^2OS is to unlock a range of unprecedented security use cases, including fine-grained dynamic security programmability of entire infrastructures, information flow tracking across an entire infrastructure, and easily translating simple, global security goals onto local OS and network policy decisions. Leveraging infrastructure-wide security control abstractions provided by S^2OS, new security applications can be easily composed to provide automatic, dynamic, host-networking coordinated, and intelligent defense.

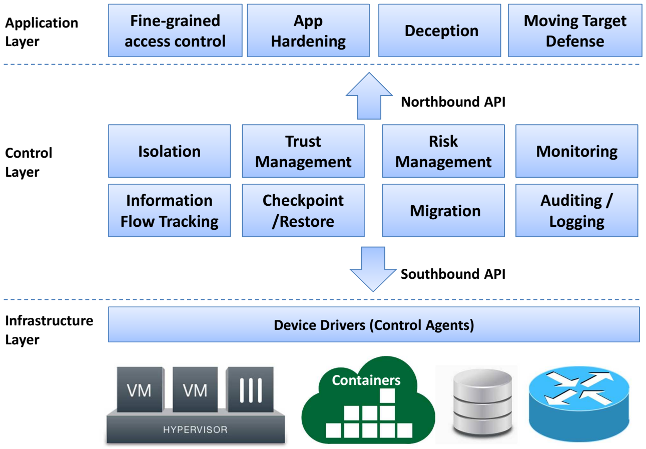

Figure 1: S^2OS overview.

2 OVERVIEW

The goal of S^2OS is to extend SDI into a comprehensive security architecture. Given the complexity of providing the programmable security across the entire SDI, S^2OS follows a layered design approach [12]. As shown in Figure 1, S^2OS contains three layers from bottom to top: *infrastructure layer*, *controller layer*, and *application layer*. Each layer implements abstractions and interfaces for the next layer up, and uses the interfaces of the layer below to perform its own functions.

The *infrastructure layer* is similar to hardware resources and drivers in a traditional OS. The S^2OS *infrastructure layer* consists of control agents in various computing and networking units in the infrastructure. We integrate control agents into hypervisors and container engines to provide control over virtual machines and containers or library OSes, respectively. More specifically, We extend the OpenFlow Extension Framework (OFX) [43] to enable OpenFlow network devices with custom functions. We also extend Open vSwitch (OVS) to design the control agents for regular hosts and mobile devices with application and context awareness.

The *controller layer* is a core layer in S^2OS that enables programming the underlying infrastructure for security applications and services. The controller layer abstracts the entire infrastructure into a logical entity. The controller layer also provides a set of security capability abstractions, such as isolation, trust management, risk management, information flow tracking, monitoring, checkpoint/restore/migration, and auditing/logging, which make it possible to implement more generic security services, and all forms of security policy management to meet security objectives. For example, S^2OS makes it easy to define and enforce consistent security policies across both hosts and networks on the entire infrastructure.

S^2OS further abstracts security capabilities as microservices [14], which are *sharable*, *reusable*, *customizable* and *scalable*, and can be automatically provisioned and dynamically migrated based on real-time security requirements. Based on composable security microservices, security applications are implemented as software instance that can be quickly instantiated and elastically scaled to deal with attack traffic variations toward flexible and on-demand placement of security functions.

In the *application layer*, S^{2}OS provides a scripting language to facilitate developing new security applications. Similar to a Unix-style pipeline, application layer scripts implement global policies using specific capabilities provided by the controller layer. Using a FRESCO-like [41] script interface to compose underlying modules, we envision that a variety of novel infrastructure-wide programmable security applications/services, such as fine-grained access control, app hardening, deception, and moving target defense, can be created on top of S^{2}OS.

In summary, S^{2}OS makes it possible to program security in the entire infrastructure through intelligent orchestration and dynamic provisioning of security services. S^{2}OS provides APIs to promote programming security applications and services with composable microservices, on-demand resource allocation, and self-service provisioning Thus, security applications and services can operate on an abstraction of the entire infrastructure, leveraging security services and capabilities without being tied to specific system implementations.

3 S^{2}OS DESIGN

This section describes each layer of the S^{2}OS design, and how these layers abstract different levels of policy concerns, ultimately building programming abstractions for datacenter-scale policies.

3.1 Infrastructure Layer

The infrastructure layer serves to virtualize low-level data center components: namely the *host* and *networking*. The goal of the infrastructure layer is to create common abstractions of heterogeneous infrastructure, to then be used as building blocks for larger-scale, control-layer functionality.

3.1.1 Host Building Blocks. One core element in SDI is an *application* (or *app* for brevity), which can be one or more processes, generally executing a single logical task. We model the infrastructure using a data flow graph, where each *app* is a node, and OS abstractions (e.g., files, IPC, network sockets) form connections, as well as sources and sinks. For example, a file on a file system can be a source for data, it flows through a file handle into an *app*, and an external network connection forms a sink. Because most OS abstractions can be bi-directional, we refer to terminal nodes in the graph as *endpoints*.

We model an SDI instance a *group* of applications. The simplest use of a group is to create a virtual instance of an operating system, with the abstraction of shared storage and a shared virtual LAN. We do not attempt complete transparency when resources or applications are placed on different physical machines, as many distributed OSes have in the past. Rather, each *app* could have a private *OS instance* with a private IP address. Applications can explicitly request to share an OS instance if they need shared OS abstractions beyond a shared file system, such as IPC. We note, and explain below, that all of these abstractions may be virtualized—an OS instance may be running in a virtual machine, a container, or a collection of cooperating library OSes; storage may be virtualized across servers; and the network is also software-defined and possibly multiplexed.

For more complex security goals and deployment models, we also allow *nested* application groups. Nested groups are a building block for drawing an explicit security perimeter around applications with different degrees of trust. As a simple example, one may wish to try out a new application inside of a sandbox. The new application can be placed inside of a nested group, and any input or output from the group will be checked by a *perimeter reference monitor*, which is responsible for mediating information flows across group boundaries, as well as sanitizing, declassifying, logging, or monitoring inputs and outputs. In other words, the perimeter reference monitor is a key "pinch point" for security policy enforcement. In the sandboxing example, one can gradually adjust the degree of suspicion and access mitigation inside the sandboxed group, as well as migrate other resources into the group.

Providing Process Virtualization. The first building block for S^{2}OS on a local OS or hypervisor is basic process virtualization support. One goal is to remain independent of a particular technology, but, rather, to adopt a generic approach that can plug a number of technologies, including virtual machines, containers, and library OSes. We propose to extend the Open Container Interface (OCI), currently promoted by Docker and other companies, as a generic abstraction for an isolated OS view.

At a high level, OCI can support any virtualization technology that can do basic tasks, such as start a virtual environment, pause execution, and migrate execution to another physical machine. When designing S^{2}OS, in addition to the well-established VM based virtualization technology, we also focus concretely on two other specific technologies: Linux Containers and the Graphene Library OS [45]. Containers offer a measure of efficiency and ease of deployment, especially in concert with an application packaging tool like Docker. Graphene is selected because it is lightweight and facilitates easy deployment of POSIX applications on new platforms, such as Intel's SGX enclave environment.

Providing Configurable Process-level Virtualization. The Graphene library OS already has a model of multiple library OS instances collaborating to provide shared OS abstractions, as well as a basic ability to dynamically isolate applications. The current model assumes all collaborating library OSes are equally trusting of one another, and primarily considers disconnection as the failure mode. We propose here to make this model more robust: with a security analysis of the risks of a misbehaving library OS instance and make resulting fixes, handle re-connection, partial connections, and apply additional scrutiny to questionable remote procedure calls. Graphene also only uses local IPC; we propose to use the well-supported 9P protocol from Plan 9 [34] as a building block for inter-host procedure calls.

In the case of containers, a significant amount of the needed infrastructure is already present to create isolated or virtualized views of host abstractions. The main missing component is the ability to bridge abstractions across multiple hosts. We propose to also leverage the same RPC protocols from Graphene within the Linux container implementation, to create virtual OS abstractions as needed. In general, our goal is to develop a substrate for OS-level virtualization that is independent of the particular virtualization technology.

3.1.2 Networking Building Blocks. While the flow-based, match-action abstraction is adequate for a conventional network data plane with packet forwarding as its main functionality (e.g., OpenFlow switches), such an abstraction is not sufficient for a far richer data plane of a software-defined infrastructure that incorporates complex network security functions. The operations on data packets required by diverse network functions will go beyond simply looking up and matching certain header fields, perhaps rewriting some of them or inserting new header fields. Thus, the challenge here is how to design new abstractions that go beyond todays simple flow-based match-action data plane abstraction to support complex network functions.

Extending OVS w/ App and Context Awareness on Hosts and Mobile Devices. For regular hosts and mobile devices in a typical enterprise setting, they may not have an underlying hypervisor. However, having observed that many of them can actually have Open vSwitch (OVS) or similar mechanisms support in their OS kernel (e.g., Linux machines and Android devices contain OVS), we can extend OVS to abstract the host/device into a virtual data plane in order to provide application-flow management (i.e., which application generates which flow in what host context, which is finer-grained than simple network flows).

We illustrate a parallel between the existing SDN data plane switch and our abstraction, which embraces the concept of a "virtual" switch on host/device. While an SDN data plane facilitates communication amongst a set of network devices via a port-host mapping, a "virtual" switch provides communication between *virtual ports* and software-entities. To enable application-flow management, we treat all host/device applications and network interfaces (e.g., WiFi,3G/4G) as network port entities on a virtual switch. By mapping applications and network interfaces to unique virtual ports, we enable flow management of all application network traffic inside of our virtual switch. This allows for easy flow management and efficient application flow isolation, as well the utilization of the SDN concepts, which readily function in data plane switches. Furthermore, we couple device context information, such as time and GPS location, with each application network flow. In this way our controller is granted access to not only application network flow information but also the *context* of the application during network activity. This enables the controller to perform advanced decision making with fine-grained connection and context information on a per-app basis for each manged host/device.

3.2 Controller Layer

The objective of the control layer is to provide a set of basic security management capabilities (or primitives). With these capabilities, various security applications can be developed, such as fine-grained access control, isolated (sandboxed) execution, moving targeted defense, and system wide information flow tracking. More specifically, we aim to provide the following capabilities (the list will be expanded over time).

Capability 1: Isolation. The ability to isolate components is a cornerstone for security. Untrusted apps must be confined and should not tamper or interfere with any other apps. An untrusted OS also should not sneak or tamper with the execution of trusted apps. We can provide isolation at the following different levels.

- **Virtual machines**. When an OS is trusted, we can execute the untrusted app in a virtual machine (VM) environment that runs the trusted OS. The security isolation is accomplished by the virtual address space separation managed by the OS and hardware. The access control mechanism provided by the OS also helps isolate the app. However, inside the VM, an untrusted app may attack other apps when the app privilege is configured inappropriately even though OS is trusted, and we therefore need another isolation mechanism—containers.
- **Containers**. A container is a process level isolation mechanism. Containers often share the same underlying OS, ideally to execute multiple instances of the same applications, Logically, the app running inside the container cannot tamper with other apps running in other containers provided that the underlying OS is trusted. However, when the OS is compromised, all containers become untrusted.
- **Secure enclaves**. Recently, with the need of running outsourced computing in untrusted platforms, there is a growing interest of designing secure enclaves to protect apps against the underlying untrusted (malicious) OS with hardware assistance. Intel SGX [4] represents such a trend, and Haven [8] and Graphene [46] demonstrate the practicality of running legacy apps with a library OS running inside SGX enclaves.

With different levels of trustworthiness, we have to provide the different isolation mechanisms. We propose to provide a programmable interface for security administrators to appropriately configure the isolation and confine the app execution across a range of trusted or untrusted host components, from traditional standard VMs, to containers, or enclaves. We propose to use these mechanisms to ensure that apps execute with *least privilege*.

Capability 2: Trust Management. Most security problems hinge on issues of trust, and current systems would benefit from tools for managing trust relationships. An *app* downloaded from untrusted sources is certainly untrusted, and we have to confine its execution in order to prevent damage to other applications and the underlying system. Meanwhile, a trusted app that consumes untrusted input is also untrusted, since untrusted input can possibly compromise the program. In addition, an app uploaded to outsourced environment (e.g., cloud) cannot trust the platform, since infrastructure owners might steal sensitive data or tamper with a library on the system. Clearly, there is a lack of mutual attestation in existing platforms. With SDI, we can introduce a trust measurement interface between the running *app* and the underlying hosting platforms, and provide mutual attestation between a virtual platform and the *app*.

Capability 3: Risk Management. Isolation is a core security abstraction for S^2OS. Keeping unrelated users or applications separate is an essential foundation for end-to-end security. However, there are a number of real-world scenarios where one may want to think about isolation as a non-binary property—weighing risk and giving limited access to a new application, perhaps during a period of careful monitoring.

Consider running a web service within an SDI that handles sensitive data. Suppose a developer wants to try out a new, freeware key-value store that boasts improved efficiency. The key-value store is probably benign, but could include malware. On current infrastructure, this scenario presents several significant challenges. First, it is difficult to test a new component of a large cloud application without replicating all of the infrastructure. Second, it may be hard to do meaningful evaluation without realistic data. Simple isolation is not sufficient any more; we need a system abstractions for managing inherent risks.

We propose to investigate the applicability of an old model—the "clans and chiefs" microkernel model [28], for drawing risk boundaries—but apply this approach to modeling risk. Essentially, the idea is to place a reference monitor at each sub-group boundary, that is tasked with risk mitigation. Risk mitigation can take several forms. For instance, before introducing a new application into the system, one can use SDI to take an end-to-end snapshot of the current storage contents and configurations—giving the ability to roll back if things go terribly wrong. Similarly, one might want to anonymize sensitive data from storage at the application group boundary, or enforce egress restrictions while monitoring the new system component.

Capability 4: Monitoring (Introspection). Supervising app execution and monitoring its behavior has been proved to be a practical approach for real-world security systems. However, existing security systems often focus on monitoring for specific attacks at specific parts of the stack. Attackers can often evade the detection by designing more advanced attacks such as interface bypassing, layer-below attacks, or network evasion. The opportunity for SDI is that we can observe network activities, each application's state, and also the kernel state, including those invisible ones hidden by attackers, giving global visibility into system state and unlocking opportunities for inference of attacks at the whole-system level.

Note that there are established introspection approaches for VM based virtualization (e.g., using kernel data structures [5, 29, 33, 38], kernel agent [17, 19, 32, 40, 44], or sibling VMs [13, 15, 16, 39]). However, these works focus on understanding what the running OS is doing, rather than into running processes. With our process virtualization, we need to support introspecting the process from the infrastructure layer. We propose to extend the library OS with an introspection built-in support, to inspect the process state out of the containers.

Capability 5: Information Flow Tracking (IFT). Another key capability in S^{2}OS is tracking information flows. Tracking information flow across the entire SDI creates a complete view for security auditing, detecting data breaches, and enforcing information-flow-based access control policies (e.g., no "write down"). While there are prior works on information flow tracking within a single host (e.g., taint analysis [10, 31], or process coloring [25]), systematically tracking the information flow across the entire infrastructure is under-explored [51].

Although label creep has been a long-standing challenge for IFT systems, we believe that tracking information flows can be useful in a number of practical scenarios. First, there are scenarios where protecting against exfiltration of sensitive data that should not be widely touched. For instance, preventing a crytographic signing key from leaving a given machine can be useful, as most applications should not handle this data. Second, there are cases where IFT policies can be too strict, but this data can be useful for security audits (i.e., "why is this label flowing through this component?"), or for tuning placement of data on logical components onto physical resources (i.e., noticing heavy movement of data with a given label over a link could lead to a migration for efficiency). In other words, IFT can be useful for analysis and inference about system behavior, even if automatically making access control decisions would be too strict. Thus, although IFT-based security is challenging, we believe there are several practical benefits to integrating IFT into end-to-end analysis and management.

Capability 6: Checkpoint, Restore, and Migration. One advantage of SDI is the ability to easily checkpoint app state and restore it on another machine. However, different host systems may not support all virtualization techniques. For instance, one may not be able to checkpoint a Linux container instance and run it on a Windows host, which could support a library OS or a hardware virtual machine. Thus, we propose to define and support a more generic checkpoint/restore mechanism. As a concrete starting point, we will take a checkpoint of a running Linux process (including from a Linux container), and then load the checkpoint into the Graphene library OS. Similarly, we propose to convert Graphene checkpoints into a format that can be reconstructed into a Linux process.

By providing the checkpoint, restore, and migration primitives, we can offer new app level protection mechanisms such as app evacuation, which evacuates a critical app from a compromised host by placing it in a "life boat" host where the app can continue to execute without disruption; app cloning, which clones the app process – possibly with binary-level randomization — in other hosts which can further be migrated to other physical hosts to maximize its survivability.

Capability 7: Auditing and Logging. Similar to many existing systems that provide logging capabilities for security audit, S^{2}OS also offers a logging primitive to provide documentary evidence of the system activities. Since there are already log facilities in existing systems, the logging in S^{2}OS will primarily focus on the logging of the primitive executions and the executions controlled by them, such as how isolation is invoked, how the isolation protected app is performed (e.g., its system call behavior), and how IFT is tracked.

On the other hand, the sheer volume of data to be generated and subsequently queried in S^{2}OS will be substantial. Simply logging is not sufficient, as policy decisions may depend on previous actions; histories on different machines must be merged, and queries must also be efficient. We propose to leverage BetrFS [24, 50]—a local Linux file system that can ingest small writes up to two orders of magnitude faster than conventional file systems (e.g., ext4 or ZFS), while preserving efficient queries. In addition to a performance profile well-suited for ingesting log data, BetrFS has additional properties that make it attractive for this purpose, such as it supports multi-version concurrency control, and has significant flexibility in scheduling I/O and logically integrating information after the fact.

3.3 Application Layer

3.3.1 Application Development Script Language. To facilitate security application development, we will leverage our previous work FRESCO [41], which provides modular, composable security services for SDN. In particular, we will use a similar concept of module-composing programming, as motivated by Click [27]. Here, a module can be a (set of) microservice(s) provided by the S^{2}OS controller, or user-defined functions. A security function/service running at the S^{2}OS application layer is realized through an assemblage of modules. Each module will define interface such as (i) input, (ii) output, (iii) parameter, (iv) action, and (v) event. As their names imply, input and output represent the interfaces that receive and transmit values for the module. A parameter is used to define the modules configuration or initialization values. A module can also define an action to implement a specific operation on network packets or flows. An event is used to notify a module when it is time to perform an action. To configure modules through a FRESCO-like script, developers must first create an instance of a module, and this instance information is defined in type variable. Developers can specify a scripts input and output, and register events for it to process by defining the scripts input, output, parameter, and event variables. Defining an instance is made very similar to defining a function in C/C++.

Based on FRESCO, we will investigate how to extend the module interfaces as well as extend more modules strategically to entire SDI in this project. We will also extend the FRESCO script language to support more rich conditional handling in SDI, in particularly including *host-based* security capabilities in addition to network-based functions, to better assist developers in composing various security functions from elementary modules.

3.3.2 Example Security Applications. With the security capabilities provided at the S^{2}OS control layer, administrators can develop security applications, similar to how developers write apps using OS APIs. Below we list several example security applications that S^{2}OS unlocks.

Fine-grained Access Control. In traditional cyberinfrastructure, system-level access control and network-level access control only work individually, without coordination across resources. However, S^{2}OS enables infrastructure-wide, end-to-end fine-grained access control. With the previously-mentioned capabilities (namely, trust, isolation, monitoring, and information flow tracking) offered by the S^{2}OS control layer, a security administrator can easily develop a fine-grained access control app to confine the workload execution, and monitor its behavior and even the information flows. More specifically, based on the trust level of the to be protected app and its underlying OS, the security administrator can specify which isolation mechanism (VMs, containers, enclaves, or network) atop which the app will be executing, what kind of host the app can talk to, and how the information can flow to other hosts/apps. Then, the isolation, monitoring, and information flow tracking capabilities will work together at the S^{2}OS control layer, to transparently supervise the app execution and enforce the security policies.

Application Hardening. One of the particular benefits of S^{2}OS is the ability to more easily create a hardened perimeter (or "shell") around a "soft" application. A significant amount of application policy enforcement will follow naturally from the limits of the SDI itself in S^{2}OS. In other words, what an application can access is already limited by SDI rules. However, one may wish to take advantage of other security features, such as logging, sandboxing, or deception/honeypot, which requires additional policy writing.

Existing OS-level security modules, such as SELinux or AppArmor, require fairly involved profiles or policies that are difficult to write. Most users of these hardening mechanisms essentially use default policies written by a third party for common applications, and, for uncommon applications, select a highly-permissive default. One essential aspect of usability is providing tools that make these policies easy for a security administrator to write.

We expect that we can write a set of policy libraries and templates that will suffice for most applications, with example templates such as system call logging, information flow tracking, and deception. The main challenges of specifying these policies arises when one needs idiosyncratic restrictions, such as an application-specific filter on specific types of network outputs. For applications on the "tail" of uncommon functionality, the developer or administrator may need to write some policy code. Nonetheless, our overarching goal is to keep this effort minimal, and commensurate with the complexity of the policy goal.

Deception. Since software inevitably contains exploitable 0-day vulnerabilities, we need a mechanism to catch these new attacks. Honeypots have been practical for this purpose. With the capabilities/primitives provided by our control layer, security administrators can easily develop light-weight, adaptive, high-interaction, software honeypots for attacker deception, disinformation, monitoring, and analysis. In contrast to traditional honeypots, which offer only weak interactivity and are therefore easily detectable by advanced persistent threats (APTs), we can build a software honeypot that arms live, commodity server software with deceptive attack-response capabilities. Under our deception framework, detected attacks (detected at either network or host level) are transparently migrated/redirected to isolated decoy environments that possess the full interactive power of the targeted victim app, but misinform adversaries with honey-data and aggressively monitor adversarial behaviors.

Moving Target Defense. With the isolation, checkpoint, restore, and migration capabilities, security administrators can use them to achieve unprecedented level of integrity, security, and resilience for the execution of mission-critical apps. Specifically, we can develop a virtualization-based attack resilient execution environment that can turn a mission-critical app into a moving target that is not statically coupled with one runtime (including the OS and underlying hardware, as well as network configurations/environments). Such an attack resilient execution environment provides strong isolation, mobility, and resilience at process granularity. The isolation property means that an assured app will not be negatively affected by other non-assured apps and their runtime; the migration property allows the assured app to dynamically move into and out of a host; and the resilience property guarantees that the assured app can actively avoid malicious apps and operating systems.

4 RELATED WORK

SDN/NFV-based Network Security. Many recent efforts have been devoted to addressing various security challenges in SDNs. The Resonance [**?**] architecture enables dynamic access control and monitoring in SDN environments. FloodGuard [48] provides a more generic DoS attack prevention extension that is not limited to TCP traffic. SPHINX [**?**] presents a novel model representation, called flow-graph, to detect several network attacks against SDN networks. TopoGuard [21] is a new solution to defend against topology poisoning attacks in SDN. FRESCO [**?**] is an SDN/OpenFlow security application development framework designed to facilitate the rapid design of SDN-enabled detection and mitigation modules. FortNOX [37] and SE-FloodLight [36]) are security constraint enforcement kernels for SDN/OpenFlow controllers. FlowGuard [22] is a framework for building robust SDN firewalls to protect OpenFlow-based networks. AvantGuard [42] advocates for the use of lightweight network security functions to enable scalable and vigilant switch flow management and defend against data-to-control-plane saturation attacks in SDNs. PBS [20] is a new solution to provide SDN-based programmable network security in BYOD (Bring Your Own Device) devices and networks.

Some recent research efforts have used NFV and SDN techniques to address the inflexibility and inelasticity limitations of traditional network defense mechanisms [11, 49**?**]. Bohatei [**?**] is a flexible virtual DDoS defense system for effective DDoS attack defense. VFW Controller [11] is virtual firewall controller that enables safe, efficient and cost-effective virtual firewall elasticity control. PSI [49] is a new enterprise network security architecture that enables fine-grained and dynamic security postures for different network devices.

In contrast, S^{2}OS is a comprehensive framework that provides security control over various enterprise/cloud/data-center computing/network units, including both fine-grained application process executions (through control agents at Hypervisor/Container/Library OS) and application-aware network flows (through control agents at extended Open vSwitch software and OpenFlow switches).

Virtualization-based Systems Security. As a layer that runs in between the hardware and OS layers, the concept hypervisor was first proposed in the 1960s [35]. In addition to pushing our computing paradigm from multi-tasking to multi-OS, hypervisors have also pushed security mechanism (e.g., monitoring) from traditional in-VM to out-of-VM, thereby achieving strong isolation. This is because guest OSes run on the virtual resources [6] that a VMM provides, which gives new opportunities for flexibility and control since VMM is essentially a software layer and software is easier to modify, migrate, and monitor. Through extracting and reconstructing the guest OS states at the VMM layer, out-of-VM security mechanisms become possible, empowering them to control, isolate, interpose, inspect, secure and manage a VM from the outside [9, 18]. Numerous research has been carried out in developing various security solutions with virtulization over the past decade, as summarized in [7, 23].

5 CONCLUSION AND FUTURE WORK

We have presented the design of S^{2}OS, a security OS designed for security and management of disparate resources, ranging from processes to storage to networking. S^{2}OS will offer an easy-to-use and programmable security model for monitoring and dynamically securing applications. We anticipate S^{2}OS could unlock a wide range of unprecedented security opportunities, including fine-grained, dynamic security programmability at infrastructure scale, and information flow tracking across an entire infrastructure.

S^{2}OS is an ambitious but decomposable, scalable and incrementally implementable architecture. We are actively developing S^{2}OS currently. The development involves a blend of modular design practice, theoretical analysis, implementation and experimentation. To move towards system prototypes, we plan to take a three-step approach. We will first construct a small-scale S^{2}OS at one institution. Once successfully tested, we will deploy it at other participating institutions and then carry out cross-institution integration and testing. The third step is to work with Internet2 [1] and CloudLab [3] resources, to validate S^{2}OS concepts and methods over the Internet2 and CloudLab platform.

Finally, balancing security and performance is an important future direction. Existing research in SDN security (e.g., [20, 42]) already showed some good promise of reasonable performance in the software-defined networking architecture. As our future work, we will investigate new techniques to achieve good tradeoffs between security and performance while designing and implementing S^{2}OS.

ACKNOWLEDGMENTS

We thank our shepherd Ahmed Ali-Eldin and the anonymous reviewers for their insightful comments. This work was supported in party by the National Science Foundation (NSF-CNS-1700527, NSF-CNS-1700544, NSF-CNS-1700499, NSF-CNS-1700507, and NSF-CNS-1700512) and VMware.

REFERENCES

[1] 1996. Internet2. (1996). https://www.internet2.edu/.

[2] 2012. Network Function Virtualisation - Introductory White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf. (2012).

[3] 2014. CloudLab. (2014). https://www.cloudlab.us/.

[4] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13.

[5] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. 2008. Automatic Inference and Enforcement of Kernel Data Structure Invariants. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC '08)*. IEEE Computer Society, Washington, DC, USA, 77–86. https://doi.org/10.1109/ACSAC.2008.29

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*. 164–177. https://doi.org/10.1145/945445.945462

[7] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. 2015. A Survey on Hypervisor Based Monitoring: Approaches, Applications, and Evolutions. *Comput. Surveys* 48, 1, Article 10 (Aug. 2015), 33 pages.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[9] Peter M. Chen and Brian D. Noble. 2001. When Virtual Is Better Than Real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HOTOS '01)*. 133–. http://dl.acm.org/citation.cfm?id=874075.876409

[10] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. 2006. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.* 3, 4 (2006), 359–389.

[11] Juan Deng, Hongda Li, Hongxin Hu, Kuang-Ching Wang, Gail-Joon Ahn, Ziming Zhao, and Wonkyu Han. 2017. On the Safety and Efficiency of Virtual Firewall Elasticity Control. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS'17)*.

[12] Edsger W. Dijkstra. 1968. The structure of the THE-multiprogramming system. *Commun. ACM* 11 (May 1968), 341–346. Issue 5. https://doi.org/10.1145/357980.357999

[13] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*. Oakland, CA, USA, 297–312.

[14] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2016. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036* (2016).

[15] Yangchun Fu and Zhiqiang Lin. 2012. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of 33rd IEEE Symposium on Security and Privacy*.

[16] Yangchun Fu and Zhiqiang Lin. 2013. Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. *ACM Trans. Inf. Syst. Secur.* 16, 2 (2013). https://doi.org/10.1145/2505124

[17] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. 2014. HYPERSHELL: A Practical Hypervisor Layer Guest OS Shell for Automated in-VM Management. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 85–96. http://dl.acm.org/citation.cfm?id=2643634.2643644

[18] Tal Garfinkel and Mendel Rosenblum. 2003. A virtual machine introspection based architecture for intrusion detection. In *Proceedings Network and Distributed Systems Security Symposium*.

[19] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. 2011. Process Implanting: A New Active Introspection Framework for Virtualization. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems (SRDS '11)*. 147–156. https://doi.org/10.1109/SRDS.2011.26

[20] Sungmin Hong, Robert Baykov, Lei Xu, Srinath Nadimpalli, and Guofei Gu. 2016. Towards SDN-Defined Programmable BYOD (Bring Your Own Device) Security. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS'16)*.

[21] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. 2015. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Proceedings of 2015 Annual Network and Distributed System Security Symposium (NDSS'15)*.

[22] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. 2014. Flow-Guard: building robust firewalls for software-defined networks. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*. ACM, 97–102.

[23] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. 2014. Sok: Introspections on trust and the semantic gap. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 605–620.

[24] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *FAST*. 301–315.

[25] Xuxian Jiang, AAron Walters, Dongyan Xu, Eugene H Spafford, Florian Buchholz, and Yi-Min Wang. 2006. Provenance-aware tracing ofworm break-in and contaminations: A process coloring approach. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE, 38–38.

[26] G Kandiraju, Hubertus Franke, MD Williams, Malgorzata Steinder, and SM Black. 2014. Software defined infrastructures. *IBM Journal of Research and Development* 58, 2/3 (2014), 2–1.

[27] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. 2000. The Click Modular Router. *ACM Transactions on Computer Systems* (August 2000).

[28] Jochen Liedtke. 1992. Clans & Chiefs. In *Architektur von Rechensystemen, 12. GI/ITG-Fachtagung*. 294–305.

[29] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. 2011. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*. San Diego, CA. http://www.isoc.org/isoc/conferences/ndss/11/pdf/3_3.pdf

[30] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. 2010. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2010), 69–74.

[31] James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'05)*. San Diego, CA.

[32] Bryan D. Payne, Martim Carbone, Monirul I. Sharif, and Wenke Lee. 2008. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of 2008 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 233–247. https://doi.org/10.1109/SP.2008.24

[33] Nick L. Petroni, Jr. and Michael Hicks. 2007. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS '07)*. ACM, 103–115. https://doi.org/10.1145/1315245.1315260

[34] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. 1990. Plan 9 from Bell Labs. In *In Proceedings of the Summer 1990 UKUUG Conference*. 1–9.

[35] Gerald J. Popek and Robert P. Goldberg. 1974. Formal requirements for virtualizable third generation architectures. *Commun. ACM* 17, 7 (1974), 412–421. https://doi.org/10.1145/361011.361073

[36] P.A. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran. 2015. Securing the Software-Defined Network Control Layer. In *In Proceedings of the ISOC Network and Distributed System Security Conference (NDSS)*.

[37] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. 2012. A Security Enforcement Kernel for OpenFlow Networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN '12)*. ACM.

[38] Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2009. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys '09)*. 47–60. https://doi.org/10.1145/1519065.1519072

[39] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. 2014. Hybrid-Bridge: Efficiently Bridging the Semantic-Gap in Virtual Machine Introspection via Decoupled Execution and Training Memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. San Diego, CA.

[40] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. 2009. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS '09)*. 477–487. https://doi.org/10.1145/1653662.1653720

[41] Seungwon Shin, Phil Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. 2013. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*.

[42] Seungwon Shin, Vinod Yegneswaran, Phil Porras, and Guofei Gu. 2013. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS13)*.

[43] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. 2016. Enabling Practical Software-defined Networking Security Applications with OFX. In *Network and Distributed System Security Symposium (NDSS)*.

[44] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. 2011. Process out-grafting: an efficient "out-of-VM" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS'11)*. ACM, Chicago, Illinois, USA, 363–374. https://doi.org/10.1145/2046707.2046751

[45] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *EuroSys*. 9:1–9:14.

[46] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX*.

[47] Tore Ulversoy. 2010. Software defined radio: Challenges and opportunities. *IEEE Communications Surveys & Tutorials* 12, 4 (2010), 531–550.

[48] Haopei Wang, Lei Xu, and Guofei Gu. 2015. FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'15)*.

[49] Tianlong Yu, Seyed K Fayaz, Michael Collins, Vyas Sekar, and Srinivasan Seshan. 2017. PSI: Precise Security Instrumentation for Enterprise Networks. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS'17)*.

[50] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2016. Optimizing Every Operation in a Write-Optimized File System. In *FAST*.

[51] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. 2008. Securing distributed systems with information flow control. In *NSDI*.