Contextual, Flow-Based Access Control with Scalable Host-based SDN Techniques

Curtis R. Taylor, Douglas C. MacFarland, Doran R. Smestad, Craig A. Shue
Worcester Polytechnic Institute
{crtaylor, dcmacfarland, doransmestad, cshue}@cs.wpi.edu

***Abstract*—Network operators can better understand their networks when armed with a detailed understanding of the network traffic and host activities. Software-defined networking (SDN) techniques have the potential to improve enterprise security, but the current techniques have well-known data plane scalability concerns and limited visibility into the host's operating context.**

In this work, we provide both detailed host-based context and fine-grained control of network flows by shifting the SDN agent functionality from the network infrastructure into the end-hosts. We allow network operators to write detailed network policy that can discriminate based on user and program information associated with network flows. In doing so, we find our approach scales far beyond the capabilities of OpenFlow switching hardware, allowing each host to create over 25 new flows per second with no practical bound on the number of established flows in the network.

I. Introduction

By fully understanding the network activity from end-systems, network operators can mitigate security risks, such as data exfiltration, the spread of malware or system compromises. In traditional systems, network operators are typically blind to intra-subnet traffic, since hosts directly forward the traffic without traversing security enforcement and monitoring devices. Recent innovations, such as the software-defined networking (SDN) paradigm, hold the potential to partially address the problem: with appropriately crafted fine-grained flows, the OpenFlow protocol [27], a widespread standard in the SDN community, allows a centralized controller to learn each time a new network flow is created.

While SDN approaches hold promise, they face two significant challenges: 1) fine-grained flows in OpenFlow's data plane controls do not scale to large networks [11] and 2) OpenFlow is inherently blind to end-host activities, since it operates in switches and routers. Studies on OpenFlow-compatible switches show that some switches are only able to handle 150 new flows per second while others handle 750 flows per second [40]. Other work has found that some commonly-used switches have high-speed TCAM memory limits of 2000-4000 entries and that, in some cases, memory swapping between TCAM and slower-speed memory can reduce the switch's new flow capacity to only 12 flows per second [24]. These switch limitations can induce performance bottlenecks and denial-of-service conditions even with benign traffic; in adversarial conditions, adversaries can easily induce switch thrashing to create network outages [36].

Beyond scalability concerns, OpenFlow does not provide network operators with detailed visibility into the end-hosts operating on the network. The OpenFlow standard creates matches based on network headers, but this information may not be semantically meaningful. As an example, a popular video conferencing application uses ports 80 and 443 for communication [29], even though these ports are intended for HTTP or HTTPS traffic. Without application layer proxies or deep packet inspection tools, operators cannot determine the actual origin or destination of the traffic or correctly determine if the traffic is communication between a Web browser and Web server. Even more concerning, malware can take a similar approach to create connections that look like Web requests while actually communicating to exfiltrate information or for command and control [8]. Network operators need details about the host context surrounding the network request to make informed access control decisions.

In this work, we ask two research questions: 1) *How can we scalably obtain flow-level information for all network traffic?* and 2) *How can we provide network operators with detailed context surrounding each network flow?*

To answer these questions, we embrace the "dumb network, smart hosts" stance. We take the OpenFlow agent functionality out of network switches and routers and instead place equivalent functionality in the end-hosts themselves, as shown in Figure 1. In doing so, we create an SDN approach that provides detailed host context and can scale to large networks while still yielding high performance.

Our contributions are the following:

- **Host context for all network flows:** We allow operators to craft detailed policies for flow authorization that include information about the applications creating the traffic. The approach is modular and allows the communication of arbitrary context. As an example, we created a policy that tracked applications and users and allowed only root-installed programs to access the network. We found that even this simple policy would successfully block multiple malware attack vectors while introducing low performance overheads.
- **Scalable, fine-grained flow-based access control:** We address the "southbound" or data-plane scalability concerns in OpenFlow by leveraging the distributed computing power of the end-hosts to apply the rules that hardware switches would otherwise be required to manage. This allows the hosts to apply fine-grain rules while allowing the hardware to apply coarse-grain rules, providing scalable, detailed network understanding. Even

in our unoptimized setup, we found that hosts could create 25 new flows per second and established flows introduced no new constraints on the hosts, scaling far beyond the capacity of TCAMs in modern OpenFlow hardware. Further, the approach introduced only 38 ms of delay to flow establishment.

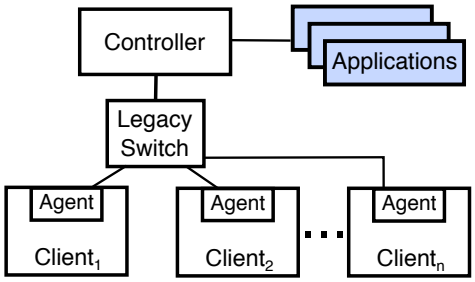

Fig. 1: Integrating SDN functionality in host-based agents allows use of legacy switch infrastructure.

In creating this approach, we note that it can be deployed immediately and inexpensively using standard enterprise software deployment tools [30]. This allows organizations to incrementally adopt the approach with minimal effort and no capital costs.

II. Related Work

We briefly describe OpenFlow, work related to fine-grained flow scalability, and research surrounding host-context in SDNs.

A. OpenFlow Fine-Grained Flow Scalability

OpenFlow allows the specification of fine-grained matching criteria, but the use of fine-grained matching introduces considerable scalability concerns. Curtis *et al.* [11] found that commodity OpenFlow switches are not built to handle the number of fine-grained flows organizations see today. As a result, they encourage aggressive use of broad, wild-carded OpenFlow rules when possible. This choice trades network visibility and control benefits for performance and scalability. Other work has shown that some OpenFlow switches can support between 750 and 2,000 flows, with an unspecified number of flows being stored in software tables [22]. Even top-end OpenFlow switches, such as the IBM RackSwitch G8264 switch [20], have a maximum of 97,000 OpenFlow rules [16]. The inability to handle fine-grained flows scalability makes OpenFlow switches a target for denial of service attacks [36].

Wang *et al.* [40] noted the poor performance of OpenFlow hardware switches with fine-grained flows. They proposed a tunneling solution that redirected traffic to distributed virtual switches (called Open vSwitch) running in computing hypervisors. Our approach is similar in that it also uses SDN agents in host software to achieve scalability. However, our approach differs in that our SDN agent runs within the host operating system itself, rather than in a hypervisor. This distinction allows us to 1) obtain detailed context about the host's environment and 2) support end-hosts that do not use virtualization, such as client machines.

B. Extracting Context from End-Hosts

Ethane [9], an early SDN implementation, sought to enhance network security by allowing network operators to write detailed security policy that could include named entities such as users, end-host machines, and access points. Unfortunately, Ethane is a switch-based SDN approach, like OpenFlow, and it lacks information from the end-hosts that is needed to enforce the policy about users. Our work embraces the ideals of Ethane and augments it by instrumenting end-hosts and providing controls that allow policy enforcement using named entities, such as users and applications.

HoNe [14] provides process attribution by correlating network traffic to processes. The approach lacks centralized coordination and does not support arbitrary host context or embrace the SDN paradigm. Dixon *et al.* [12] use virtual machines and TPMs to allow network administrators to securely push network management to the end-hosts themselves but they lack situational awareness inherent to OpenFlow based SDNs. Parno *et al.* [34] present an approach called Assayer that uses end-host TPM capabilities to explore performance and security aspects of networks where the end-host verifies state already being maintained locally (e.g., number of packets sent) rather than requiring another device to determine the state manually. Participating systems push policies to off-path verifiers that supply clients with tokens to allow continual communication. Assayer's approach does not follow the OpenFlow SDN model, is reactive, and does not scale when attestation is required on a per packet basis.

Naous *et al.* [32] proposed a revision to the `ident` [21] protocol to allow a remote system to query for details about the application and other information associated with a flow. The authors designed `ident++` to work under the OpenFlow protocol to allow network operators to delegate administration of end-hosts from a centralized operator to local operators in the network. Our work shares the goal of fusing end-host information with network control with `ident++`. However, `ident++` does not describe or evaluate an implementation of the approach nor does it indicate how it would overcome the inherent scalability concerns of fine-grained flows in a switch-centric SDN architecture. In our approach, we take a host-based approach to address scalability. We then create and evaluate an implementation of the approach, both using a native OS solution and using a bump-in-the-wire implementation.

Other approaches have focused more on the context available on an end-host and how to extract this information for automated systems to better understand a user's workflow. These works can augment our approach and range from collecting mouse-clicks and keyboard presses [10] to application-specific implementations such as the user's interaction with a web browser [26], [41]. Each of these approaches can be used to inform the host-based agents in our architecture, providing more context on the system's operation and enable stronger policies to be written.

III. Threat Model: User-Level Adversary

We deliberately scope our threat model to yield tangible results to many organizations in common scenarios while describing avenues to relax the stronger assumptions. In our threat model, we consider an external adversary that has compromised a user-level account on a system inside the defending organization's perimeter. The following are our two key assumptions.

- **Trusted Operating System:** Most host-based defenses, including anti-virus software, software firewalls, and host intrusion detection software assume that a system compromise only occurs at the regular user level, consistent with the best practice of "least user privilege" [35], [38]. We share this assumption, but note it may be relaxed using techniques such as trusted computing hardware or virtualization with trusted hypervisors. Even without such innovations, our approach can directly address many common user-level compromise attacks.
- **No Physical Attacks:** We focus on an adversary that lacks a physical presence inside the organization; otherwise, an adversary could sabotage systems and or use custom hardware to bypass our implementation. While we may address physical attackers in the future, we note that many attacks are launched remotely.

Since our approach instruments end-hosts, we focus on devices that can be modified by an organization's IT staff. For legacy devices (e.g., network printers) or "bring your own device" equipment, organizations can use individual VLANs to isolate the devices and proxy all the device traffic through a trusted network forwarding system. This approach allows full flow-management compatibility for these devices, albeit with a performance overhead.

IV. Approach: SDN via Host Agents

Given OpenFlow's scalability concerns and lack of host context, we instead take a host-based approach. We push all of the fine-grained rule matching and control to system-level software agents running on each of the hosts. The network infrastructure may continue using coarse-grain rules, whether in a legacy enterprise network or in a network using OpenFlow. The approach only minimally affects end-host performance and scalability because these end-hosts already manage per-flow state to manage the connection, as in TCP connections.

In describing our approach, we provide details of our reference implementation on the Ubuntu Linux operating system. While the details of the approach will vary across operating systems, the concepts are consistent and similar functionality may be available. To enable communication between agents and the controller, we used asynchronous messaging with the Twisted framework [39]. These components were not optimized for performance and thus are conservative estimates of what would be possible in a production implementation.

While the approach is intuitive, it achieves powerful outcomes. Our system not only replicates the elevation and caching paradigm of traditional OpenFlow, it further supports actions analogous to the "actions" in OpenFlow [2]. Some details of the behaviors may differ slightly as OpenFlow resides within network switches capable of controlling the datalink layer. Being a host-based implementation, our approach natively works at the network layer and above, though we do have the ability to influence datalink layer actions. We note that in addition to replicating the OpenFlow functionality, our approach scales even to extremely large networks as the end-hosts only store entries for their own flows and the logically centralized controller can be physically implemented using distributed controllers [37].

In addition to flow-based controls and context, our approach has the following features:

- A capability to arbitrarily route traffic through proxies, IDSes, and other middleboxes,
- A modular design allowing arbitrary plug-ins to enable additional host context on demand,
- Explicit notification to network controllers when a flow ends, allowing accurate real-time network flow insight,
- Optimal traffic filtering at the source host to avoid network overheads, and
- Avoids the need for kernel or application modifications by using established kernel features.

We now describe how we achieve each of these outcomes.

A. Host Agent: Intercepting Packets

Our host agent does not require special kernel or application modifications. However, the agent does run with administrator privileges, allowing it to manage the system's configuration and operation. Similar agent-based system administration tools are popular in large enterprises [30] and the agent software can be installed as a system service using traditional enterprise software deployment mechanisms. As a result, organizations can quickly and easily deploy the technology across parts or all of the enterprise network.

In our implementation, we leverage the connection marking feature of the `iptables` firewall: when a flow has been vetted, we update the marking value stored in the kernel's connection tracking table. We then use the Linux kernel's `netfilter_queue` library to tell the kernel that it should intercept any unmarked packets and send them to an agent running on the host. We further update `iptables` to create a special "drop mark" that can be used to discard all packets in connections that have that marking. Therefore, if the controller ever decides to disallow a network flow, it may command the sending host's agent to set the drop mark on the flow, causing all packets in the flow to be dropped (either silently or with an ICMP error to the sending application) before entering the network. Accordingly, a controller can squelch malicious behaviors efficiently, with no overhead or state in the network. This allows the controller to easily mitigate traffic floods.

In Figure 2, we provide an overview of how the SDN agent manages an end-host. When two communicating parties are using our approach, the process shown in Figure 2 is completed by both participants. In this process, the initial packet transmitted will not match any existing approved kernel

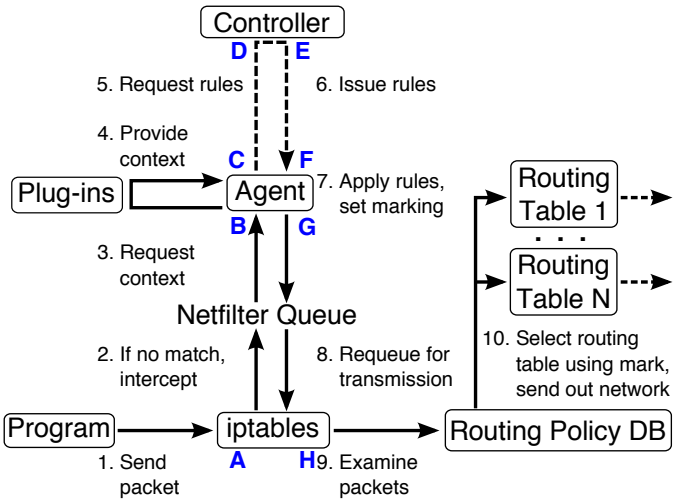

Fig. 2: Overview of kernel and agent communication. Dashed lines represent network traffic while solid lines represent intra-system communication. The bold, blue letters indicate measurement points for the performance evaluation in Table I.

flow, which is specified using the network layer addresses, transport protocol, and transport layer ports. Accordingly, while the packet is queued for transmission in the OS kernel, our SDN agent will extract the packet from the kernel queue.

Once the agent has intercepted the application's packet, it analyzes it and determines the context for the communication (Figure 2, steps 3 and 4). The agent is extensible and can encode and transmit arbitrary host context from any data source on the end-host. As an example, the agent may determine the owner and executable path associated with the process and provide this context. The agent then transmits a message to the SDN controller (Figure 2, step 5), which contains the flow tuple and the extracted host context, and requests instructions from the controller and, if desired, the packet payload as well.

Once the host agent receives a response from the SDN controller, the host agent will install appropriate NAT, firewall, routing and forwarding rules supplied by the controller (Figure 2, step 7). The agent then indicates the flow should not be diverted to the agent in the future. In our Linux implementation, we use a temporary `iptables` rule to update the marking for the flow to indicate the flow is authorized. The agent then signals `netfilter_queue` to release the packet, providing the altered version if requested by the controller.

Unlike in OpenFlow, where the controller does not directly learn when a connection ends, the host agent can inform the controller about connection terminations. In our implementation, we intercept the `CONNTRACK_DESTROY` kernel event using the `netfilter-conntrack` library and alert the controller. OpenFlow instead uses timeouts to approximate when a connection ends, causing the flow to be re-elevated to the controller if it continues. In our implementation, we use the `netfilter-cttimeout` library to re-create this functionality. Our approach allows the controller to have real-time knowledge of the network, rather than relying upon timeouts to approximate the network activity.

Normally, the controller will allow the host to forward packets using its default routing table. However, the controller may choose to specify an arbitrary next hop for the flow instead. This can be used to proxy traffic through a third-party, such as an IDS or application-layer firewall. To do so, the controller orders the host-agent to create a unique routing table for each available next hop. The table contains a single entry: a default route to the controller's desired next hop. The controller can then use policy routing to specify which flows should use the alternate routing table. In our implementation, we use the Linux `ip-rule` command to manage the routing policy database (RPDB). We create a policy rule indicating that the connection marking from the controller should be re-used to determine which routing table to use. This allows the controller to specify the default routing table or an arbitrary secondary routing table, dictating the connection's next hop behavior. Next hop hosts, and by extension the connection marks associated with each, can be reused across connections. The alternate routing tables can forward to a host inside the subnet, specify a host outside the subnet, or even indicate that traffic should be tunneled via a specified waypoint. At any time, the controller may alter or remove the forwarding instructions without interrupting the flow.

Since the host agent runs on both the initiator and responder systems, the controller will have the ability to control all network flows as long as one of the participating end-host deploys the agent. Accordingly, the approach grants operators full access control and at least partial host context for the communication. If both hosts deploy, the controller can fuse the context on the initiating and responding systems for a comprehensive view of the system.

B. Host Agent: Extracting Operating Context

Given the administrator privileges of the host agent, it can gather arbitrary information from the host and transmit it to the controller. The agent can be modified to gather whatever information is needed for the network operator to write effective policy. Accordingly, in our implementation, we used a modular design that can include any number of arbitrary plugins to provide context from the host to the controller. In our initial implementation, we built a plugin to provide information about the process associated with a specified network flow along with the owner and user group associated with the process.

Starting in v3.18, the Linux kernel's `netfilter_queue` library allows the agent to determine the user name and group associated with the extracted packet. To gather data about the process using the socket, we use an approach similar to `lsof`. Once we obtain the process ID associated with the socket, we extract additional details from `/proc`, including the executable path associated with the process and the command line arguments used when the executable was launched. We examine the executable path and indicate whether any directory or file in the directory path is owned or writeable by a non-root user. We further collect similar information about all

the process's ancestors (e.g., parent process). We also collect whether a given ancestor is a shell or a GUI coordinator (such as a window manager).

Future plugins could easily extract context about the connection's flow rate and number of bytes transferred or other system features, such as resource activity (e.g., CPU load, memory consumption, disk I/O) or integrate with SELinux policy and containers.

C. SDN Controller

While our work focuses on modifying end-hosts to provide greater context to the network controller, the controller itself is an important consideration. In the future, we plan to make implementation enhancements for improving performance. However, our current implementation is a Python controller that interprets fairly simple policy and pushes rules to the host agents.

V. Security Enhancements: Contextual Policies

The increased visibility and control inherent in the fine-grained flows we enable can directly empower security systems [15], [17], [23]. Further, our approach enables new network security policies. We now describe such policies and their potential.

Network operators can use a variety of contextual languages, such as `POL-ETH` [9], Flow-based Security Language (FSL) [18], and Flow-based Management Language [19], to specify the high-level policies for a network. While these policies are amenable to formal analysis, their current instantiations are unable to distinguish among multiple users on a system. While prior work proposed such differentiation in the future [32], to our knowledge, our effort is the first to actually do so. Further, our approach provides additional contextual information from the end-host that was not considered in some of these prior efforts.

To illustrate the power and simplicity of the policies available, we provide an example for a Linux environment that was not possible to enforce in prior work and highlight the power associated with it. We express the policy in English, while noting the policy can be easily translated into programmatic conditions. The policy is written with the intention that it would be considered in order and in a short-circuited manner (i.e., the first applicable grant or deny decision is used and processing aborts without considering subsequent steps).

1) **Allow Administrative Processes:** If the process requesting network access is owned by user ID 0 through 999, grant access.
2) **Deny All User-Installed Programs:** If the process requesting network access, or any of the process's ancestors (e.g., such as its parent process), was started from an executable that was not installed by an administrator (i.e., one or more files or directories in the program's path are owned or writable by a regular user), deny access.
3) **Default Allow:** Allow network access by default.

This policy allows administrative background and daemon processes to run (rule 1) and ensures that only process from trusted, administrator-installed sources can use the network (rule 2).

This policy can act as a template that can be tailored to additional organization constraints. For example, standard network firewall policies could be inserted at the beginning of the chain, since they do not require knowledge of the host context. Application-specific constraints, such as only allowing certain Web browsers or applications with specified command-line parameters (e.g., options to disable Javascript), can be inserted between rules 2 and 3.

While this work focuses on policy incorporating host context, policy enforcement is flexible and can also contain more traditional network policies such as connection rate limiting and network isolation between hosts.

VI. Evaluation

To demonstrate and evaluate our approach, we create an implementation in a small network of virtual machines (VMs). These VMs run on a single server with 16 cores operating at 2.8 GHz and 64 GBytes running a KVM hypervisor. Each client system is allocated a single core and 512 MBytes of RAM. The network controller is allocated two cores and 2048 MBytes of RAM. All machines use Ubuntu 14.04 Server as the host operating system. For timing analysis, each host runs an NTP client and the VM server's host operating system runs an NTP server to keep the VM clocks synchronized. Each host has `iptables` preinstalled and we load the `conntrack` kernel module to allow fine-grained manipulation. The hosts are configured to ignore ICMP redirect messages, which can be generated when an intermediate hop is specified for a connection between hosts in the same subnet. Though enabled by default, ignoring such ICMP messages is a good security practice [7], [25].

To evaluate the approach, we consider the performance of the agent instrumentation, the data plane and controller scalability across the network, and the effectiveness of the security policy.

A. Host Agent Performance

When considering an SDN system, the performance of the SDN agent (the data plane) and controller (the control plane) are the key considerations. While we perform basic performance measurements of our unoptimized SDN controller, our primary contribution is enhancing the data plane. Prior work that focuses on SDN controller scalability [37] can likewise be leveraged in our approach.

The host SDN agents, and the kernel components the agents manipulate, have little impact on memory consumption, CPU, and network bandwidth (which we verified empirically). The approach does not introduce any new additional per-flow state, nor does it involve any computationally-intense operations. While bandwidth may initially seem to be a concern, the host-agent interception process is only involved at the beginning of a connection and only for a single round-trip. Accordingly, once the connection is established, the traffic incurs no additional bandwidth or latency overheads.

The key performance metric for our approach, and that of traditional OpenFlow, is the latency overhead associated with elevating a new flow to the controller for consideration. In our approach, we also query plug-ins for host context, which may introduce additional latency. To characterize the latency overhead, we rapidly spawn new flows on the host agents and compare the results to those in traditional OpenFlow.

For this experiment, the host context gathered consists of the user ID, primary group ID, application path, application arguments, if the process and all ancestor processes are from administrator-installed paths, and details about the environment (e.g. displayed in the foreground or run in a shell).

In Table I, we show the latency introduced by each step of the process. We see that our SDN agent incurrs a median of just under 17 milliseconds, with a significant portion of that time being devoted to gathering the host context such as the application path as discussed above. Further, this overhead is only incurred at the beginning of the network connection and thus may have little impact on actual applications since it is during the traditional connection build-up phase (in, for example, TCP's slow start).

Component Description	Fig. 2 Steps Start	End	Median (ms)	Std. Dev. (ms)
Initial Interception	A	B	0.088	0.105
Obtain Host Context	B	C	6.803	1.435
Elevation to Controller	C	F	3.535	1.688
Controller Decision	D	E	0.005	0.002
Marking	F	G	3.976	0.487
Re-queuing	G	H	0.022	0.005
Overall End-to-End	A	H	16.72	1.403

TABLE I: Component-wise characterization of latency overheads over 1,000 TCP connections. Columns 2 and 3 correspond to the bold, blue letters in Figure 2.

To better understand the performance overheads, we performed high resolution timing on the hosts. We recorded the clock timestamp at each of the locations of the elevation process indicated by the bold, blue letters in Figure 2. We performed these timings on one of the hosts and the controller using `ovs-benchmark`'s [3] batch mode to create $1,000$ sequential connections. For each connection, the policy presented in Section V was enforced based on the context gathered on the end-host. To avoid introducing inaccuracies from nested timings, we conducted additional trials for the timings of the overall end-to-end timings with all intermediate timing samples disabled. We present the results of the timing experiment in Table I.

From the timing experiment, we can see that the communication between the kernel and our agent via `netfilter_queue` takes minimal time, as does the decision on the controller. Only three steps caused more than 100 microseconds of delay: the gathering of host context, the round-trip to the controller, and the packet marking approach. Fortunately, there is significant room to optimize each of these components. The host context collection can be parallelized, the communication protocol can be greatly simplified, and the packet marking can use a more efficient `netfilter-conntrack` call rather than forking a process to invoke the `iptables` executable. Further, the use of a compiled language rather than Python would likely greatly improve performance.

Num. Hosts	New Flows/s	Median RTT (ms)	Std. Dev. (ms)
2	27.4	34	9.48
4	26.7	36	7.46
6	26.0	38	5.52
8	25.5	39	5.86
10	25.1	39	6.09
12	24.6	40	6.67
14	23.2	41	8.26
28	12.4	78	19.93

TABLE II: Round trip times with each host transmitting 1,000 packets.

B. Scalability of the Controller and Agents

In a second set of experiments, we explore the scalability of our approach with the rapid creation of new flows. In these experiments, we vary the number of communicating hosts from two machines up to fourteen, adding two machines each trial, and run one additional experiment using 28 hosts. Using `ovs-benchmark`'s batch mode, each host sequentially creates $1,000$ new TCP flows to another host. Each hosts sends and receives the same number of requests to ensure no host is more overburdened than another. The host receiving a connection request is not configured to listen for connections and responds with a `TCP+RST` to allow the sender to quickly calculate the RTT. Both the TCP request and response are elevated to the controller for approval as previously described. We record the number of new flows per second that a single host could create. We run an additional experiment with 28 hosts to confirm that our testing infrastructure is limited by the number of cores on the hosting server. For all experiments, each host was pinned to a single core.

We present the results of our scalability tests in Table II. As expected, the median RTT numbers are roughly double the end-to-end results from Table I because both the initiator and the responder must contact the controller for approval of the flow. In the case of 28 hosts, the over-subscribing of the CPU cores did indeed introduce timing artifacts. When considering traditional OpenFlow using an Open vSwitch to connect two hosts, the flows per second are 243.3 and the median latency is roughly 4 milliseconds. While Open vSwitch has years of development and is built using a compiled language, thus achieving better performance, it is unable to provide the context we can provide in our approach. With further optimizations, our approach may yield more competitive performance.

In our scalability tests, we induced roughly 350 flows per second (14 hosts) with each host creating approximately 25 new flows per second. This new flow rate greatly exceeds the rate in Ethane [9], which induced less than three new flows per second in the worst case. Importantly, unlike OpenFlow or other hardware switch-based SDN implementations, all the data plane flow state is stored at the hosts themselves, eliminating any network constraints on the number of established

flows. In essence, the number of flows created per second and the total number of flows a host may have are limited only by the computational resources on the host and the amount of time to vet the request at the controller. Our timing results show that our controller can handle around 200,000 new flows per second by spending around 5 μs on each packet as shown in Table 2. In practice, the connection processing overheads may decrease this value. The POX controller, which is also implemented in Python, can only handle around 35,000 packets per second [13]. Accordingly, we do not expect the examination of host context in our approach will significantly degrade the controller's scalability.

C. Evaluating Policy Enhancements on Security

In Section V, we provided an example policy for the network controller. In it, the controller will only allow regular users to create network connections if the process was created from a root-installed program (e.g., `/usr/bin/`). We now evaluate whether such a policy would be able to thwart persistent user-level malware.

We first perform an experiment using a simulated Linux malware called n00bRAT [5]. The executable provides an adversary with the ability to connect to a compromised machine and run preconfigured commands such as grabbing `/etc/passwd` and exfiltrating it. We modified the malware's source to run on a non-privileged port to match our threat model of user-level compromises. Accordingly, any commands preconfigured in the malware that require root access will be denied by the OS when attempting execution. The malware can be delivered through multiple vectors, including as an attachment in a phishing message or as a drive-by download on a vulnerable Web browser. In evaluating our policy, we test a case where a user on a host (that implements our approach) receives and runs the malware from an email attachment. We also perform a browser-based attack using Metasploit [4] and launch the malware using the compromised browser. In both cases, the malware is denied network access using our simple policy.

When executed as an attachment in a popular email application, the malware begins running as a separate process from the mail reader's attachment folder. Because the process was created by a regular user executing a user-installed program, our policy denies any connections, preventing the malware from being able to receive connections and commands from an attacker. That is, connections both originating from and destined to the malware will be denied regardless of whether the remote host is inside or outside of the protected network.

The drive-by download case is more interesting. Using Metasploit, we use the CVE-2013-1710 vulnerability in Firefox to allow a remote shell to be established with an attacker. The vulnerability allows the adversary to run arbitrary code within a new thread in the Web browser. Our policy will allow the adversary to establish a connection to download the n00bRAT malware to the user's machine, since the Firefox process is root-installed. However, if the adversary then launches the n00bRAT malware, our policy denies the malware any network access since it is not root installed. As a result, the adversary can only have connectivity with the targeted machine for the duration that Firefox executes. Other persistence strategies, such as cron-jobs or start-up scripts, will also fail since the executed malware comes from an untrusted source.

These results show that even simple network policies at the controller can significantly affect the spread of malware. With application-specific policies and greater context, defenders may be able to detect and prevent the spread of even advanced malware.

VII. Discussion

We now consider how the approach would be deployed within an organization and the functionality of hosts when remote to the organizational network.

A. Partial Deployment

By using software on end-hosts, our approach allows organizations to use standard software deployment tools to ensure each of the hosts at the organization deploy the software. At the same time, organizations may choose to deploy the approach in a piecemeal fashion, deploying to subsets of the organization by function (e.g., starting with information technology staff) or based on machine role (e.g., administrative systems before development systems). Organizations may also be constrained by the presence of user-owned devices, such as in the "bring your own device" (BYOD) approach. As we will discuss, our approach can interact well with legacy devices and embedded devices that cannot be altered.

When an organization is in a partial deployment, there are three scenarios that can arise: both hosts deploy, neither host deploys, and a mixed case where only one host deploys. The first case is the focus of the rest of the paper and can be considered equivalent to full deployment. In the case where neither host deploys, we degrade to the limitations of a traditional network infrastructure and lack insight into the traffic between the hosts. Finally, having a single host participating in a flow is analogous to an external host communicating to an internal host. In this scenario, the implementing host can still enforce any policies set forth by the controller.

Organizations may have a set of hosts that will never deploy the approach, such as network printers or embedded devices. To protect these assets, organizations may place each in an isolated VLAN containing only the single asset and a proxying device that employs the flow-level access control of an implementing host. This approach does require the proxying device to be trusted by the organization and multiple physical proxies may be required to avoid bottlenecks. Further, the approach does not gain context inside the host. While imperfect, this proxying approach does allow a deployment option to accommodate legacy and BYOD equipment without needing client-side modifications. Organizations may also consider virtualization techniques where hosts are required to use virtual infrastructure for policy enforcement [31]. Since

virtualization resides outside the host, these techniques will also lack host context.

Our ability to support partial deployment means an enterprise can strategically choose what hosts they want deploy the agent on. This is in stark contrast to OpenFlow, which requires hosts to be physically connected to the same switch and restricts the deployability process.

B. Compatibility with Non-Linux Hosts

Our initial implementation uses the Linux kernel, but it can be applied natively on other operating systems, such as Apple's Mac OS X and Microsoft's Windows OS. Mac OS X's built-in firewall, `pf`, is based upon OpenBSD's firewall implementation by the same name [33]. BSD systems provide a special socket interface, called divert sockets, which can be used to intercept packets for the host agent. Such systems provide additional support for packet tagging rules and policy based routing, which are the remaining features needed for the host agent. In Microsoft Windows, the Windows Filtering Platform [28] may provide the needed support for host agents, but further exploration is needed for conclusive results.

Some other devices or operating systems may be unable to support a native host agent. To support these devices, we created a bump-in-the-wire solution using a Raspberry Pi 1 Model B+. The device provides all the network control features of our approach. We used the device's built-in Ethernet card along with a USB NIC to forward and control traffic between a connected host and the rest of the network. We tested the device on a host running Mac OS X Yosemite and a host running Windows XP and confirmed our ability to control the traffic flows identically to a native solution. This approach allows for a plug-and-play style deployment for new devices. However, the approach only supports the network control functionality; it does not gain host context. A smaller host-based agent could be used on partially supported operating systems to gather limited host context and inform the Pi during connections.

C. Potential for Network Security Policy

In a 2013 study of vulnerabilities on the Windows platform, researchers found that 96% of the critical Windows vulnerabilities and 100% of the Internet Explorer could be eliminated by removing administrator rights from the user's account [1]. Further, malware that injects itself into processes, such as the Zeus botnet [8], will be unable to inject into long-lived system processes as a user. Instead, the malware would only be able to inject less persistent user-level processes. While browsers are an attractive target, since they typically have regular authorized access to the network, other exploits, such as malicious code in PDF or word processor documents, may be more easily thwarted by policy since those applications rarely engage in network connectivity. Even in Web browsers, policies preventing certain traffic, such as SMTP communication, can constrain the abilities of injected malware. Simply having insight into the responsible application can greatly enhance organizational network policy.

Our system also enables policies for the graceful degradation of mission-critical systems faced with a user-level compromise. Organizations must make strategic choices about dealing with compromised hosts on their networks. From a security perspective, it may be appealing to immediately remediate any compromised systems and restore from backups. This can compete with the desire to preserve forensics for prosecution or for counter-intelligence [6]. In other cases, organizations may have practical constraints that hinder remediation efforts, such as running mission-critical services on the machine, essential on-going data collection, or even a simply constrained support staff for the organization. Unfortunately, in traditional networks, the choices can be rather limited: 1) isolate the host or 2) allow the host to communicate arbitrarily. The former approach may hinder mission continuity while the latter approach may introduce unacceptable risks for an organization.

In our approach, we enable fine-grained policies with host context by default, allowing organizations to have flexibility in responding to a compromise. Rather than fully isolate the system, an organization may choose to only allow a known client application on the machine to talk to a whitelisted set of applications on specific servers in the organization. This policy would be enforced on the compromised system and all other hosts in the network. This provides robust control, including intra-subnet traffic, with minimal disruption to the network and systems while tightly constraining access. Such a specific policy can yield tighter controls than approaches such as OpenFlow or network firewalls, with less risk of collateral damage, by leveraging host-specific context.

VIII. Concluding Remarks

Our novel SDN agent approach provides scalable flow-based monitoring for enterprise networks. With it, organizations can reuse their existing network infrastructure and incrementally deploy the approach. With logically-centralized access controllers, operators can understand the context of the network request, such as the application being used and the username of the user. This enables richer and more powerful organizational network policy.

We created a prototype implementation and evaluated in a real physical network with diverse systems. We evaluated the approach at a higher scale using a virtual network. In doing so, we found that our approach incurred minimal overheads. Our work provides a foundation for potential future work. We will explore proxying solutions, both physical and virtual, for legacy devices or assets not owned by the organization. We will also examine bursty network traffic and how virtualization and trusted computing technology can be leveraged to relax some requirements in our trust model. Finally, we will explore the policy potential for enterprise network systems and how such policy may be complemented using SDN hardware.

Acknowledgments

The authors would like to thank Robert Hollinger for his help and feedback on an earlier implementation of this

approach.

This material is based upon work supported by the National Science Foundation under Grant No. 1422180. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] "2013 microsoft vulnerabilities study: Mitigating risk by removing user privileges," in *Avetco whitepaper*, 2013.

[2] "Openflow specification version 1.3.0," June 2014.

[3] "Open vswitch manual," http://openvswitch.org/support/dist-docs/ovs-benchmark.1.txt, April 2015.

[4] "Penetration Testing Software — Metasploit," http://www.metasploit.com, April 2015.

[5] "Remote Administration Toolkit (or Trojan) for POSiX (Linux/Unix) system working as a Web Service," https://github.com/abhishekkr/n00bRAT, April 2015.

[6] F. Adelstein, "Live forensics: diagnosing your system without killing it first," *Communications of the ACM*, vol. 49, no. 2, pp. 63–66, 2006.

[7] S. M. Bellovin, "Security problems in the tcp/ip protocol suite," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 2, pp. 32–48, 1989.

[8] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang, "On the analysis of the zeus botnet crimeware toolkit," in *Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on*. IEEE, 2010, pp. 31–38.

[9] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 1–12, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1282427.1282382

[10] W. Cui, R. H. Katz, and W.-t. Tan, "Design and implementation of an extrusion-based break-in detector for personal computers," in *Computer Security Applications Conference, 21st Annual*. IEEE, 2005, pp. 10–pp.

[11] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018466

[12] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy, "Ettm: A scalable fault tolerant network manager," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 85–98.

[13] D. Erickson, "The beacon openflow controller," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 13–18.

[14] G. A. Fink, V. Duggirala, R. Correa, and C. North, "Bridging the host-network divide: Survey, taxonomy, and solution," in *Proceedings of the 20th Conference on Large Installation System Administration*, ser. LISA '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 20–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267793.1267813

[15] C. Gates, M. Collins, M. Duggan, A. Kompanek, and M. Thomas, "More netflow tools for performance and security," in *Proceedings of the 18th USENIX Conference on System Administration*, ser. LISA '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 121–132. [Online]. Available: http://dl.acm.org/citation.cfm?id=1052676.1052691

[16] GEANT, "Technology investigation of openflow and testing," GEANT Whitepaper DJ1-2.1. [Online] http://geant3.archive.geant.net/Media_Centre/Media_Library/Media%20Library/GN3-13-003_DJ1-2-1_Technology-Investigation-of-OpenFlow-and-Testing.pdf, July 2013.

[17] K. Gennuso, "Shedding light on security incidents using network flows," http://www.sans.org/reading-room/whitepapers/incident/shedding-light-security-incidents-network-flows-33935, 2012.

[18] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker, "Expressing and enforcing flow-based network security policies," 2008.

[19] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, ser. WREN '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1592681.1592683

[20] IBM Corporation, "Ibm system networking RackSwitch G8264," IBM Data Sheet [Online] http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?subtype=SP&infotype=PM&appname=STGE_QC_QC_USEN&htmlfid=QCD03020USEN&attachment=QCD03020USEN.PDF#loaded, July 2014.

[21] M. S. Johns, "Identification protocol," IETF RFC 1413, February 1993.

[22] M. Kuzniar, P. Peresini, and D. Kostic, "What you need to know about sdn flow tables," in *Lecture Notes in Computer Science (LNCS)*, no. EPFL-CONF-204742, 2015.

[23] K. Lakkaraju, W. Yurcik, and A. J. Lee, "NVisionIP: Netflow visualizations of system state for security situational awareness," in *ACM Workshop on Visualization and Data Mining for Computer Security*, October 2004.

[24] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu, "Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization," in *ACM International on Conference on Emerging Networking Experiments and Technologiesmerging Networking Experiments and Technologies*. ACM, 2014, pp. 199–212.

[25] C. Low, "Icmp attacks illustrated," *SANS Institute URL: http://rr. sans. org/threats/ICMP_attacks. php (12/11/2001)*, 2001.

[26] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, "Blade: an attack-agnostic approach for preventing drive-by malware infections," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 440–450.

[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[28] Microsoft, "Windows filter platform architecture overview," https://msdn.microsoft.com/en-us/library/windows/hardware/ff571066(v=vs.85).aspx.

[29] ——, "Which ports need to be open to use skype for windows desktop?" Skype FAQ 148 https://support.skype.com/en/faq/FA148/which-ports-need-to-be-open-to-use-skype-for-windows-desktop, April 2015.

[30] Mircosoft, "System center configuration manager," http://technet.microsoft.com/en-us/systemcenter/bb507744.aspx, 2014.

[31] D. Montero, M. Yannuzzi, A. Shaw, L. Jacquin, A. Pastor, R. Serral-Gracia, A. Lioy, F. Risso, C. Basile, R. Sassu, M. Nemirovsky, F. Ciaccia, M. Georgiades, S. Charalambides, J. Kuusijarvi, and F. Bosco, "Virtualized security at the network edge: a user-centric approach," *Communications Magazine, IEEE*, vol. 53, no. 4, pp. 176–186, April 2015.

[32] J. Naous, R. Stutsman, D. Mazieres, N. McKeown, and N. Zeldovich, "Delegating network security with more information," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*. ACM, 2009, pp. 19–26.

[33] OpenBSD Developers, "Pf: The openbsd packet filter," http://www.openbsd.org/faq/pf/.

[34] B. Parno, Z. Zhou, and A. Perrig, "Using trustworthy host-based information in the network," in *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, ser. STC '12. New York, NY, USA: ACM, 2012, pp. 33–44.

[35] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," Proceedings of the IEEE, 1975.

[36] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "Sdn security: A survey," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, Nov 2013, pp. 1–7.

[37] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, ser. INM/WREN'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863133.1863136

[38] M. Tracy, W. Jansen, K. Scarfone, and T. Winogard, "Guidelines on securing public web servers," *NIST Special Publication 800-44, Version 2*, p. 142, September 2007.

[39] Twisted Framework Developers, "Python twisted framework," https://twistedmatrix.com/.

[40] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen, "Scotch: Elastically scaling up SDN control-plane using vswitch based overlay," in *ACM International on Conference on Emerging Networking Experiments and Technologies*. ACM, 2014, pp. 403–414.

[41] H. Zhang, W. Banick, D. Yao, and N. Ramakrishnan, "User intention-based traffic dependence analysis for anomaly detection," in *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*. IEEE, 2012, pp. 104–112.