Controller-Oblivious Dynamic Access Control in Software-Defined Networks

Steven R. Gomez*, Samuel Jero*, Richard Skowyra*, Jason Martin†, Patrick Sullivan*, David Bigelow†,
Zachary Ellenbogen*, Bryan C. Ward*, Hamed Okhravi* and James W. Landry†
MIT Lincoln Laboratory, Lexington, MA USA
email: *{*first.last*}@ll.mit.edu, †{jnmartin, dbigelow, jwlandry}@ll.mit.edu

***Abstract*—Conventional network access control approaches are static (*e.g.,* user roles in Active Directory), coarse-grained (*e.g.,* 802.1x), or both (*e.g.,* VLANs). Such systems are unable to meaningfully stop or hinder motivated attackers seeking to spread throughout an enterprise network. To address this threat, we present Dynamic Flow Isolation (DFI), a novel architecture for supporting dynamic, fine-grained access control policies enforced in a Software-Defined Network (SDN). These policies can emit and revoke specific access control rules automatically in response to network events like users logging off, letting the network adaptively reduce unnecessary reachability that could be potentially leveraged by attackers. DFI is oblivious to the SDN controller implementation and processes new packets prior to the controller, making DFI's access control resilient to a malicious or faulty controller or its applications. We implemented DFI for OpenFlow networks and demonstrated it on an enterprise SDN testbed with around 100 end hosts and servers. Finally, we evaluated the performance of DFI and how it enables a novel policy, which is otherwise difficult to enforce, that protects against a surrogate of the recent NotPetya malware in an infection scenario. We found that the threat was most limited in its ability to spread using our policy, which automatically restricted network flows over the course of the attack, compared to no access control or a static role-based policy.**

I. Introduction

Access control in traditional enterprise networks is challenging to implement in a fine-grained manner because the Ethernet and IP protocols are architected to enable connectivity rather than restrict it. Static, coarse-grained access control can be implemented at Layer 2 via VLANs or 802.11x [1], but more fine-grained or dynamic approaches must generally be implemented at the application layer (*e.g.,* Kerberos). This has two consequences that can be taken advantage of by malicious parties. First, enterprise networks have a high degree of reachability between machines in Layers 2 and 3, even if application-layer traffic is unauthorized (discussed further in Section II). This reachability, which may be unnecessary for the mission of the network, enables adversaries to exploit many software vulnerabilities (*e.g.,* in the network stack or authentication logic) regardless of their ability to authenticate. Second, handling access control at the application layer introduces complexity with respect to credential management, especially when considering edge cases. For example, Active Directory (AD) credentials are cached locally on endpoints so a user on a machine disconnected from the network can still log on locally. Unfortunately, attackers with system-level privileges can dump these credentials and use them to authenticate remotely as the victim, even if that victim is not legitimately logged onto any devices.

Both techniques are common steps in attacks against enterprise networks that involve lateral movement, in which the attacker spreads from an initial foothold deeper into the network in order to compromise a high-value machine, such as a database containing personally-identifiable information. Recently, the NotPetya family of ransomware used both of the above techniques to infect over one million computers in Ukraine in 2017, spanning two thousand companies and causing over $10 billion in total damages [2], [3]. It gained a foothold inside an enterprise via a compromised update server, then spread to other systems using a combination of vulnerability exploitation and credential theft [4]. Similar patterns were used recently in high-profile attacks against Equifax [5], Bangladesh Bank [6], Anthem [7], Chase [8], Target [9], and RSA Security [10]. Increased threats from ransomware and the resurgence of self-propagating worms in 2018 [11] further highlight the importance of the lateral-movement threat.

It is clear that static, coarse-grained Layer 2 access control systems do not effectively inhibit an attacker's ability to reach target machines, and fine-grained application-layer approaches can be bypassed through exploitation. Furthermore, neither is sufficiently aware of the larger context of user activities to distinguish between legitimate user log-on events and malicious credential theft. In order to secure enterprise networks, we propose three requirements for an access control system. First, it must enforce permissions that are at least as fine-grained as existing application-layer approaches (*e.g.,* user- and machine-specific roles). Second, it must enforce permissions in the network infrastructure to prevent endpoint software exploitation. Third, it must support policies that grant or revoke fine-grained permissions in response to security-relevant events happening in-network or on network endpoints.

A number of challenges must be overcome to support such a fine-grained, dynamic network access control system. First, defining event-driven policies on high-level identifiers (*e.g.,*

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited. This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

usernames and hostnames) but enforcing them in-network requires the ability to dynamically map these identifiers to potentially changing packet-header information (*e.g.,* IP and MAC address). Second, the complete access control policy cannot be fully cached in switches because of switch memory limitations and policy fragments that cannot always be mapped to concrete flow-rule match fields. For example, policy about a user cannot be specified in terms of IP addresses when the user is logged off all devices. Third, in order to achieve policy-switch consistency, as events arrive and cause policy changes, any new policy must be instantiated in network switches and now-stale policies must be evicted. This consistency must be achieved without disrupting ongoing network flows that remain allowed, or enabling flows that are newly denied by policy. Fourth, the system must not be easily bypassable by an attacker who has compromised network endpoints, even if they can send arbitrary traffic into the network. Finally, this system must be able to operate on large networks without imposing prohibitive delay on allowed network flows.

To address these challenges, we present an OpenFlow-based architecture called Dynamic Flow Isolation (DFI) for controller-oblivious, dynamic network access control. This paper makes the following contributions:

- The design of a novel architecture that enables dynamic network access control policies independently of the OpenFlow controller, enabling a variety of control-plane configurations and providing protection from malicious SDN controller applications.
- An implementation of the architecture that demonstrates its feasibility and overcomes the above-mentioned design challenges.
- A quantitative evaluation of the system's overhead showing that DFI increases the time-to-first-byte latency for data transiting an SDN by 17.8ms under no load. This additional latency increases to 86.7ms at 700 flows/sec, when saturation begins. The maximum throughput DFI can achieve is approximately 1350 flows/sec.
- A demonstration of an attack scenario in which DFI enforces a novel, dynamic role-based access control policy that is uniquely enabled by the system. We evaluate DFI using a surrogate of the NotPetya self-propagating malware, and show that the DFI policy slows and limits the surrogate's spread.

II. Dynamic Access Control Using SDNs

Intranetwork access control in traditional Ethernet IP enterprise networks has historically been both static and coarse-grained. For example, 802.1x [1] is a port-based access-control standard that conducts a single authentication check when a device connects to the network. Based on this check, all traffic to that device is either allowed or denied. 802.1x is static: events occurring in the network post-authentication (*e.g.,* users logging off) cannot influence the policy implemented earlier. It is also coarse-grained, since network flows cannot be allowed or denied individually. This is a limitation of the architecture of Ethernet IP networks. All devices in the same collision domain are accessible to one another at Layer 2, and thus can mutually send and receive network flows regardless of higher-layer policies. VLANs allow finer-grained control of Layer 2 endpoint reachability, but the set of reachable hosts remains static after configuration. Switches are configured to add or remove specific VLAN tags on specific ports regardless of the actual network usage by the device on that port. At Layer 3, routing tables are relatively static and cannot be changed at machine timescales to tailor network reachability across subnets for particular devices.

A growing recognition of the need for more dynamic, fine-grained control, combined with the limitations above, has prompted attempts to retrofit this capability at the application layer. Google's BeyondCorp [12] system, for example, places every network service behind an authentication proxy that authenticates the user and device based on a variety of security sensors. Unfortunately, it is not clear how this approach interacts with pre-authentication vulnerabilities, such as those in a TCP/IP stack triggered upon packet receipt [13], that can lead to kernel compromise. This approach also requires the addition of network middleware whose performance impacts have not been reported.

A more foundational approach to network access control has been taken by the academic community. This approach leverages Software-Defined Networking (SDN), specifically the OpenFlow architecture. OpenFlow divides the network into physically separate control and data planes. The *data plane* forwards traffic from endpoints based on flow rules installed in OpenFlow switches. These flow rules consist of two parts: a pattern to match against packet header fields, and an action to take in the event of a match, such as forwarding over an egress port. The *control plane* contains a logically-centralized controller that reprograms the flow tables in each switch in response to packets forwarded from data plane switches, for which there are no matching flow rules. These are referred to as `Packet-in` events, and consist of an OpenFlow message containing the packet header and associated metadata, such as the switch and port on which the packet was received. By inspecting `Packet-in` events (and other OpenFlow messages sent from the switches), the controller can reactively add and remove flow rules that together determine the logical network topology. This allows the network to adapt over time, based on endpoint traffic and the controller's network-management logic.

The majority of modern SDN controllers today, including Floodlight [14], ONOS [15], and OpenDaylight [16], provide a firewall application that can be used to implement fine-grained access control within the network, rather than being limited to the perimeter like a traditional firewall. Unfortunately, firewall rules defined using such applications are static: the policy they enforce does not change in response to events or other changing security context in the network (*e.g.,* user logs on).

Several academic prototypes have sought to implement fine-grained access control that modifies the network based on a sensed security context. Amman and Sommer [17] provide an API by which the Bro IDS can actuate network access control

policies in response to detection events, for example. Unfortunately, this approach is purely reactive. Access control rules are installed only after a malicious action has been detected by Bro, making false positives and negatives a concern. Kinetic [18] takes a different approach by providing a custom SDN controller that supports interaction through a flexible policy language. Arbitrary event sources can drive policy decisions, enabling access control that is both dynamic and fine-grained. However, Kinetic's architecture has not been shown to scale to large enterprise systems. It is also unclear how network devices with changing identifiers can be reliably specified in Kinetic policies, as these policies are not updated when network identifier-state changes.

Dynamic, fine-grained access control is a key defensive capability when considering modern threats to enterprise networks such as insider attacks and advanced persistent threats (APTs). Existing attempts to implement it leveraging SDNs have had partial success, but no one system has been both scalable for enterprises and able to provide a framework for the development of event-driven access control policies. Next we consider the challenges for developing a system like this and how they can be addressed.

III. Design of Dynamic Flow Isolation

In this section, we present the design of a novel architecture for enforcing event-driven access control policies in software defined networks, which we call Dynamic Flow Isolation (DFI). We first discuss design challenges related to managing dynamic identifiers and policies efficiently, then describe the system architecture and how it addresses these challenges. Finally, we present an end-to-end example illustrating the operation of DFI.

A. Design Challenges

High-Level Identifiers in Event-Driven Policies. A network access-control policy rule should be specified at a high enough level that it can be understood by network administrators and easily expressed by policy authors. This is particularly important in a dynamic access control system, since an administrator must be able to understand the current policy, characterize policies, and debug policy conflicts. Thus, we want to enable writing policy over high-level identifiers like hostnames and usernames that are often more human-readable and memorable compared to identifiers like MAC addresses and IP addresses. However, network devices fundamentally only filter packets based on the identifiers present in the actual network traffic. In particular, OpenFlow-based SDNs support filtering based on fields in the Link (Ethernet), Network (IP), and Transport (TCP/UDP) layers.

As a result, there is a semantic gap between the high-level entity identifiers, like hostnames and usernames, that we would like to define policy over, and the enforcement of these policies (flow rules) in the network. In order to handle policies written with high-level identifiers, these identifiers must be mapped to the correct set of low-level identifiers that hardware can use to enforce the policy, *e.g.,* MAC and IP addresses. At the same time, mappings between high- and low-level identifiers change regularly. Consider a wired host that moves from one physical network port to another, a wireless host moving between access points, or dynamic DNS mappings between a hostname and IP addresses. Any mechanism used to map between high- and low-level identifiers must maintain correctness in the face of these changes.

Caching Policy in Switches. Managing the installation and caching of flow rules in switches is challenging when the set of policy rules is large and changes over time. Proactively installing flow rules that implement policies is not efficient when the set of policies is large, because hardware switches can only store a limited number of rules, usually in the range of 512 to 8,192 [19]–[22]. There may simply not be space for all policy and routing rules for all possible allowed flows. Therefore, the subset of rules to install in the switches must be chosen intelligently.

Furthermore, policy rules may contain high-level identifiers that cannot be resolved to low-level identifiers until a later time. For example, the policy rule "The device with hostname `h1` is not allowed to send to TCP port 22 on the device with hostname `h2`" must be compiled into one written over low-level identifiers for the switches, as discussed previously. However, if `h1` or `h2` is not yet on the network, it might not have an IP address reserved. Until that happens, the policy rule cannot be compiled into a flow rule and pushed to the switch.

Policy-Switch Consistency. In an event-driven access control system, policy changes may be frequent. The system must be able to change the policy enforced on the network in a timely manner by removing stale rules that are inconsistent with the current security policy.

OpenFlow-based SDNs support two automated mechanisms for removing stale rules: hard and soft timeouts. Neither is suitable as-is for the proposed system, due to unacceptable performance or security implications.

Hard flow timeouts cause a rule to be deleted after a specified amount of time. While this helps bound how long a rule may be stale, hard timeouts interact poorly with long-running flows. If the hard timeout expires while the rule is still in active use, packets (possibly thousands in a high bandwidth flow) will be forwarded to the control plane until the rule is re-installed. Control-plane processing is orders of magnitude slower than hardware-backed forwarding, so timing-out an active flow could cause both a noticeable latency spike and impose additional load on the control plane.

Soft flow timeouts cause a rule to be deleted if a packet has not been matched against that rule for a specified amount of time. This prevents long-running flows from being interrupted. However, it also introduces a correctness issue: stale flow rules that no longer reflect current access control policy continue to reside on switches for as long as they in use. In effect, access control policy updates would not be applied to pre-existing flows.

Since neither of these mechanisms is sufficient, a consistency management system is needed that provides both timely

and efficient expiration of stale rules.

Bypass Prevention. Even the most secure access control policy is not useful if it can be circumvented easily. For that reason, it is important to design access control systems to be resistant to bypass. One area of concern in SDNs is ensuring that the controller cannot be compromised before access control checks are done. A number of recent works have demonstrated the vulnerability of controllers to various poisoning attacks on topology and network identifiers, as well as malicious apps [23]–[26]. This suggests that access control must be done before any other processing in the control plane. That way, illegal packets can be rejected before they are able to poison other controller components, possibly to bypass access control mechanisms.

Efficient Operation. Finally, in addition to handling policy updates and new flows correctly, an access control system with event-driven policies must also perform efficiently, minimizing undue latency on flows and load on the control plane.

B. DFI Architecture

At a high level, DFI consists of five components that interact with each other to decide on policies in response to events, manage current policies, resolve high-level identifiers, compile policies to flow rules, and then manage those rules in the switches. We discuss each of these components below and display the overall architecture in Figure 1.

Policy Decision Points (PDPs). The role of a PDP is to evaluate conditions that apply to a desired event-driven access control policy – for example, "When host `h1` has a log-on event, enable its network access". The PDP then *decides* whether its policy applies based on those conditions, and automatically creates or revokes rules that implement the current policy. To do this, a PDP subscribes to zero or more sensor feeds, whose events can originate from the data plane (*e.g.,* DNS, DHCP servers), end hosts (*e.g.,* anti-virus software), the control plane (*e.g.,* OpenFlow events), or even off-network (*e.g.,* a building alarm system).

DFI supports deploying multiple PDPs, enabling each to focus on providing a particular type of policy (Role-Based Access Control, Quarantine Upon Compromise, *etc.*). Conflicts between policy rules emitted by different PDPs are resolved by the Policy Manager using a unique priority assigned to the PDP by the network administrator.

Policy rules themselves are tuples consisting of *(Action, Flow Properties, Source, Destination)*. *Action* can be Allow or Deny, and *Flow Properties* include EtherType and IP protocol values. *Source* and *Destination* describe the endpoints of flows matching this rule as tuples over the following identifiers: username, hostname, IP address, TCP/UDP port, MAC address, switch port, and switch DPID. Each field can be either a specific value or a wildcard.

For example, a PDP that enforces a user-based policy might emit the following policy:

$$(\texttt{Allow}, (*, *), (\texttt{Alice}, *, *, *, *, *, *, *, *), (\texttt{Bob}, *, *, *, *, *, *, *, *))$$

This policy would permit any machine that Alice is using to communicate over any protocol with any machine that Bob is using.

If a PDP later determines that a policy rule it has generated no longer applies, it can revoke that rule using a unique identifier assigned when the policy rule was added. Revocation is distinct from installing a second policy rule with the opposite Action. After revocation, a previously-matched flow will now be matched against any other policy rules, likely from other PDPs. A PDP should not generate multiple rules that match a single flow with conflicting Action values, since rules inherit the priority value of their PDP; however, in this case, the Deny action will be used to err on the side of stopping unauthorized flows. Similarly, in the absence of any matching policy rule, DFI is configured to deny a flow by default.

Policy Manager. The Policy Manager receives policy rules and revocations from PDPs, performs consistency checks, and stores the current global policy. It also enables the Policy Compilation Point to query current policy to determine what action should be taken for a specific flow.

Consistency checks are extremely important to solving the Policy-Switch Consistency challenge mentioned earlier. When a new policy rule is inserted, the Policy Manager identifies any existing policy rules that potentially conflict with the new one, since these policy rules may have been used to add flow rules still present in the network's switches. Conflicts are possible where: 1) each flow identifier in an existing rule matches the new one (exactly or wildcarded), 2) the policy actions are different, and 3) the priority of the existing rule is lower than the priority of the new rule. The Policy Manager then tells the Policy Compilation Point to remove any flow rules derived from these conflicting policies from the switches, as described later. Conflicting policies are not removed from the policy database: this action merely flushes rules installed in the switches, ensuring that ongoing flows potentially affected by the policy change will be re-evaluated. When a policy rule *is* explicitly revoked by a PDP, the Policy Manager also instructs the Policy Compilation Point to flush any flow rules derived from this policy from the switches. In this way, flow rules are removed quickly without paying the latency and performance costs of using hard timeouts.

Entity Resolution Manager. The Entity Resolution Manager is responsible for maintaining current mappings between the high-level identifiers, like usernames and hostnames, that are used in policy rules and the actual low-level identifiers, like IP addresses and MAC addresses, that appear in network traffic and can be used in the switch's flow rules. Additionally, it prevents spoofed traffic from being able to bypass policy by ensuring that identifiers at all levels must match the expected bindings. To do this, the Entity Resolution Manager tracks the four identifier bindings shown in Figure 3, linking username $\leftrightarrow$ hostname $\leftrightarrow$ IP address $\leftrightarrow$ MAC address $\leftrightarrow$ switch and port.

Some of these bindings are many-to-many and can change over time. For instance, users may log onto multiple hosts, which may have more than one IP address associated with dif-

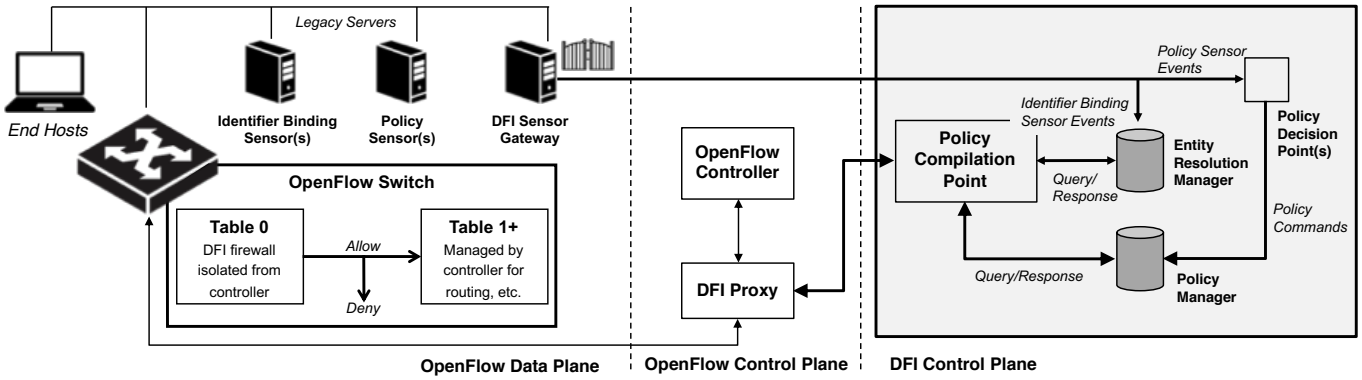


Fig. 1: Architecture of DFI

ferent network interface cards, all of which may be connected at different physical switch ports. Similarly, IP addresses may change across DHCP leases, machines may move to different physical ports, and MAC addresses change with interface (*e.g.,* wired vs. wireless).

To maintain these bindings, the Entity Resolution Manager subscribes to events from identifier-binding sensors across the network. These sensors are positioned to collect bindings from authoritative data sources. Authoritative sources are those responsible for providing one part of the binding between two identifiers. For example, DNS is the authoritative source for the binding between a hostname and an IP address, as it provides the hostname for that IP address. Hence, our sensor for the hostname to IP address binding collects those bindings directly from the DNS server. Similarly, our IP Address to MAC address sensor collects its bindings directly from the DHCP server, which is authoritative for that binding. By using authoritative data sources, we prevent attackers from poisoning the Manager with illegitimate state. Poisoning attacks from these authoritative sources are out of scope for this work, since an attacker who has compromised the sources could simply assign themselves identifiers needed to match or circumvent a target access control policy.

Using its knowledge of current identifier bindings, the Entity Resolution Manager responds to queries about new flows from the Policy Compilation Point, returning any identifiers associated with the source and destination of the queried flow. For instance, given an IP address and MAC address from a packet, the Entity Resolution Manager would return any associated hostname or username. This information can then be used to identify matching policy rules.

Instead of mapping high-level identifiers in policies to low-level identifiers when those policies are added, we choose to map low-level identifiers in packets to high-level identifiers during the access control decision. This is crucial to solve a number of challenges we discussed earlier. First, it ensures that the mappings between high-level and low-level identifiers are current at the time the access control decision is made. If policy identifiers were mapped when the policy rule is inserted, the policy rule would become incorrect as soon as any identifier bindings used in the mapping changed. Second, it enables us to write policy using identifiers that do not currently have bindings. For instance, we can write policy for a user who is not currently logged onto any machine. If identifiers are mapped when the policy rule is added, this would cause an error. However, mapping identifiers during the access control decision avoids this problem, because bindings for the user must exist (updated at the user log-on event) at the time she sends traffic.

Policy Compilation Point (PCP). The PCP is ultimately responsible for managing DFI's policy rules in the switches. In particular, the PCP processes new flow requests from switches and installs rules that apply the current policy for the flow. As mentioned earlier, the PCP also flushes flow rules from switches at the direction of the Policy Manager after a PDP decides to update a policy.

When a switch receives a packet that does not match an installed flow rule in Table 0, it forwards that to the control plane as an OpenFlow `Packet-in` event, where the DFI Proxy (detailed below) will send it to the PCP. The PCP parses the `Packet-in` event and collects all source and destination identifiers present in the packet header (*e.g.,* MAC addresses and IP addresses) as well as any information supplied by the switch (*e.g.,* in-port on which packet was received). It then queries the Entity Resolution Manager with this information to obtain any other associated identifiers, like hostname and username, and then queries the Policy Manager for policies that match this flow. The Policy Manager will return the highest-priority policy rule matching the flow, if any. Using the action specified in the policy rule, the PCP creates a flow rule specific to this flow and installs it into the switch. If no policy rule matches a new flow, DFI uses a default Deny policy. Each flow rule is built to match only the exact flow that was examined by the PCP – all available identifiers (*e.g.,* MAC addresses, IP addresses, TCP/UDP ports, *etc.*) are specified in the rule. This ensures that each new flow will be checked against current policy by DFI. We also note that allowing a packet to be forwarded at a switch results in the next switch in the flow's data path receiving it and repeating this process, so the correct policy is always applied at each hop in the flow.

The PCP also processes rule removal requests from the Policy Manager. As discussed above, these requests are issued when policy is added or removed by PDPs in order to ensure that the flow rules cached on switches remain consistent with the current policy. To accomplish this, each rule inserted into a switch is tagged with a small piece of persistent metadata—the `cookie` value in OpenFlow—indicating the policy from which it is derived. Then, the PCP uses the metadata value corresponding to the relevant policy to tell switches to flush rules derived from that policy.

While minimizing the number of flows processed is beyond the scope of this work, there is opportunity to extend DFI with a system for reactive caching of wildcarded flow rules, as in the recent CAB-ACME system [27]. A key challenge is to avoid caching wildcarded flow rules that match packets for which higher-priority policy rules may exist in the Policy Manager's database. This is non-trivial for DFI because we expect changes in the policy database over time, and these policy rules may contain identifiers that must be mapped during rule compilation for the SDN.

DFI Proxy. We design DFI to be independent of the controller to ensure that the controller and its apps cannot violate or interfere with the access control policy that DFI is enforcing, either accidentally or intentionally, solving the No-Bypass challenge mentioned earlier. To do this, we insert a proxy between the switches and the SDN controller, a technique used successfully by a variety of earlier SDN security and reliability tools [28]–[32].

This proxy is responsible for smoothly interposing DFI's access control prior to the SDN controller and its applications and is designed to avoid being a single point of failure in the architecture. The state it maintains does not persist between sessions and is relevant only to its particular switch connections. Multiple proxies, as well as PCPs, can be used in parallel in an SDN installation for reliability or performance. The proxy is designed to accomplish two primary goals: isolate rules inserted by DFI from those inserted by the controller, ensuring that rules from DFI take precedence, and route `Packet-in` messages properly.

To isolate rules inserted by DFI from those inserted by the controller, the DFI Proxy takes advantage of the multiple flow tables available in OpenFlow 1.3 and above. In particular, it reserves Table 0, the first table incoming traffic is checked against, in each switch for DFI's access control rules. This is achieved by rewriting references to tables in all OpenFlow messages. Section IV contains additional implementation details about this modification. The end result is that DFI and the SDN controller are writing rules into separate tables where DFI's rules take precedence. Since the DFI Proxy intercepts the OpenFlow connections for all switches, DFI is aware of the exact path each flow takes.

The proxy also ensures that incoming `Packet-in` events from switches are processed by DFI prior to being sent to the controller. If a packet is denied by DFI, it is not forwarded to the controller at all, ensuring that the controller is not poisoned by inconsistent network state from blocked packets.

C. End-to-End Example

To demonstrate the end-to-end operation of DFI, we present a simple example, which is depicted in Figure 2. Each endpoint is running a Security Information and Event Management (SIEM) collector, such as Splunk, that provides local authentication events to a central indexer. In addition, each endpoint is part of a Windows domain managed by an Active Directory (AD) server providing DHCP and DNS services.

Consider the policy "When Alice is logged on, the computer she is using can communicate with the email server. When she is logged off, it cannot," and the situation in which Alice logs on, checks her email, and logs off the computer. The sequence of events, denoted by the numbers in Figure 2, is presented as a linear sequence for ease of exposition. Note that in practice many of these events are concurrent with one another and will occur asynchronously.

① `Alice-Laptop` joins the AD domain and is assigned an IP address by DHCP. This traffic is permitted via default allow rules.

② The Hostname-IP and IP-MAC identifier-binding sensors connected to the DNS and DHCP services on the AD server report these identifier pairs associated with `Alice-Laptop` to the Entity Resolution Manager.

③ Alice logs on via AD.

④ The log-on/log-off sensor connected to the SIEM collector notifies the Entity Resolution Manager and Policy Decision Point that Alice has logged on to `Alice-Laptop`.

⑤ The Policy Decision Point inserts a policy rule into the Policy Manager that allows flows from Alice to the email server.

⑥ Alice tries to check her email. The first packet of the flow is sent to the control plane via a `Packet-in` event.

⑦ The DFI proxy intercepts the `Packet-in` and sends it to the Policy Compilation Point.

⑧ The Policy Compilation Point queries the Entity Resolution Manager service to enrich the source and destination IP and MAC addresses with associated hostnames and users.

⑨ The Policy Compilation Point queries the Policy Manager for policy matching the enriched source and destination identifiers. The Policy Manager returns an Allow decision for the flow.

⑩ The Policy Compilation Point creates an Allow-action flow rule that matches the packet in the `Packet-in` event and sends it to switch.

⑪ The DFI Proxy forwards the `Packet-in` to the OpenFlow controller, which installs forwarding rules in the switch.

⑫ Alice checks her email, then logs off the computer.

⑬ The log-on/log-off sensor sends a binding expiration event to the Entity Resolution Manager and a log-off event to the Policy Decision Point.

⑭ The Policy Decision Point revokes this policy, which causes the Policy Manager to notify the Policy Compilation Point that any flow rules for this policy need to be removed.

⑮ The Policy Compilation Point removes any rules associated with that policy from the switches.

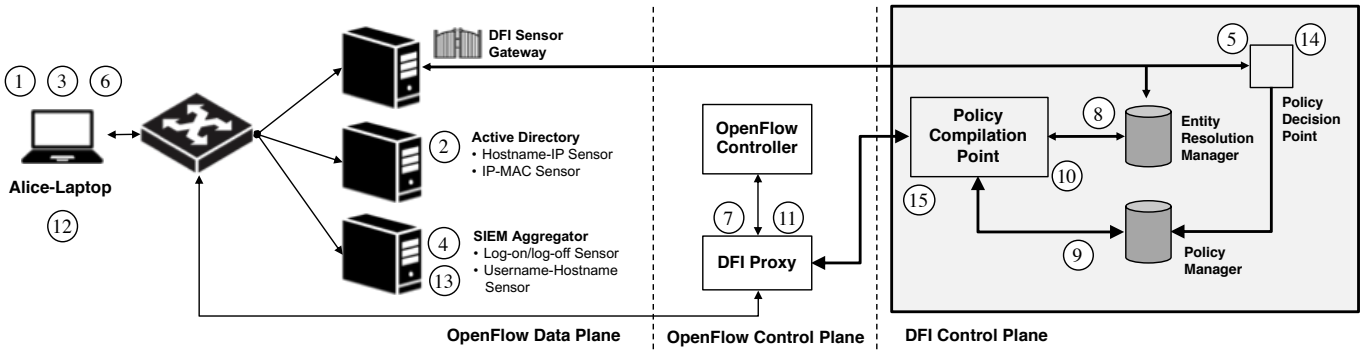


Fig. 2: DFI Workflow for Example Authentication-Based Policy

Authoritative Source	Network Identifier
System Event Logs	Username
DNS Server	Hostname
DHCP Server	IP Address
`Packet-In` Event	MAC Address
	Switch ID & Port

Fig. 3: Authoritative Sources of Identifier Bindings

IV. Implementation

DFI is implemented as a set of communicating servers providing the core functions detailed in Figure 1. These servers include one or more Policy Decision Points, a Policy Manager, an Entity Resolution Manager, and a Policy Compilation Point. These components are implemented in Java and use a RabbitMQ message bus to communicate. The messages exchanged between these components are created using protocol buffers [33] to remove language dependencies when extending or building new components for the system. Both the Policy Manager and the Entity Resolution Manager are backed by MySQL databases that maintain a record of current policy rules and current identifier bindings.

The DFI Proxy is implemented as a Java application that listens for new connections from switches; for each, it creates and manages two additional connections to the controller and PCP. The sockets may be optionally secured using TLS to encrypt all exchanged OpenFlow messages. The Proxy and PCP both use OpenFlowJ to parse these messages. Due to DFI's use of multiple flow tables, DFI supports OpenFlow versions 1.3 or later.

A. Identifier-Binding Sensors

The Entity Resolution Manager relies on binding sensors spread throughout the network to collect information on bindings and ensure that it always has the most current information. It is particularly important that this binding information always come from authoritative sources to prevent attackers from being able to poison DFI's view of the network. As a result, we implemented sensors for each of the four bindings that our Entity Resolution Manager tracks, each of which collects its bindings from their authoritative source, most of which are in the data-plane. Figure 3 shows these bindings and their authoritative source.

The MAC address to switch port binding is challenging because traditional networks maintain this binding only implicitly, via learning switches, as the last location from which a MAC address sent traffic. Since this binding is tied to the physical location of network traffic, we implement it as part of the PCP in the control plane. This sensor ensures that each MAC address is associated with at most one port on each switch, and sends updates to the Entity Resolution Manager.

The username to hostname binding is challenging for a similar reason. Active Directory (AD) and similar directory services do not keep track of users who are currently logged on, and therefore cannot be queried to obtain this information. AD grants users a Kerberos Ticket-Granting-Ticket and does not track subsequent log-on or log-off events. Local event logs maintained by each endpoint contain some of this information; however, there are multiple ways for users to authenticate in an operating system like Windows, each of which generates different events in the log. After experimenting with different approaches, our sensor implementation maintains a current count of running processes associated with a user, aggregated from endpoint logs. This is calculated by monitoring process creation and termination events. When the total number of running processes for a user on a host is greater than zero, the user is considered logged on and able to create flows from the host. When the total number is zero, they are considered logged off the system. We collect this information and centrally determine user log-on and log-off events using Splunk, a widely used Security and Information Event Management (SIEM) tool.

B. DFI Proxy

The DFI Proxy aims to enforce DFI's access control without altering the expected controller and application behavior for allowed flows. This becomes challenging because the controller assumes it has full access to the switch's tables and statistics. As a result, the DFI Proxy must transparently isolate DFI's access control rules from the controller's rules.

The proxy takes advantage of a feature added in OpenFlow 1.3 and later called flow-table pipelining. Pipelining enables

a switch to partition its memory for rules into multiple flow tables, with an incoming `Packet-in` being matched against rules in Table 0 first. A matching rule with the `goto_table` action can pass the packet to another table. This action is accompanied by a table index (`table_id`) value indicating the next table. All `Flow-Mod` messages and some others (*e.g.,* statistics requests) also contain a `table_id` denoting the table to modify or query.

Our proxy leverages this feature to reserve Table 0 for access control rules from DFI. Tables 1 and higher are reserved for the controller. If a flow is allowed, it is forwarded to Table 1, which only contains rules from the controller, for further instructions that could include forwarding or even continued pipelining into higher tables. Denied flows are dropped at Table 0. Reserving Table 0 for DFI means that the controller should not be able to modify Table 0 or learn about its contents. We implement this transparently by shifting by one all `table_id` references in messages from the controller to the switch. Similarly, any table reference being sent from the switch to the controller, *e.g.,* in a statistics reply, must also be decremented to avoid confusing the controller. This operation also ensures that existing controller applications function normally alongside DFI for flows that it allows.

V. Evaluation

A. Performance Evaluation

We first evaluate the DFI control plane in terms of microbenchmarks about its minimum latency handling a flow and its maximum throughput of new flows. We then consider how a network with DFI performs end-to-end using a small hardware SDN with OpenFlow switches and an SDN controller; here we measure the Time to First Byte (TTFB) of new flows, both with and without DFI, as a function of load on the network. Other metrics like the total DFI flow rules produced are highly dependent on policies and operational factors (*e.g.,* traffic) and therefore are not the focus of this evaluation (see Section III for options for reducing flow rules in the PCP).

The testbed for these experiments included VMs created and managed by VMware vSphere, with four 2.1 GHz Intel Xeon cores and 7.6 GB of RAM running CentOS 7. One server hosted the core DFI services (PDP, Policy Manager, Entity Resolution Manager, and PCP) while the DFI Proxy and SDN controller (ONOS 1.13) ran on another. Our data plane consisted of three end hosts and a single software switch running Open vSwitch 2.5.4.

Latency and Throughput Microbenchmarks. The flow-start latency and maximum throughput of the DFI control plane help characterize its performance independent of the SDN controller and network service. When a packet cannot be matched with an existing flow rule received on a switch (usually at the start of a new flow), it is sent to be handled by the DFI control plane, incurring some computation time before returning an access control rule for the packet. Once flow rules are installed, subsequent packets in the flow match these rules and are routed directly through the data plane without additional latency. The maximum throughput of new flows represents the level of network activity beyond which new flows will experience disconnections or extreme delays.

TABLE I: DFI Performance Microbenchmarks

Metric	Mean ± Std. Dev.
Latency (under no load)	5.73ms ± 3.39ms
Throughput (at saturation)	1350 flows/sec ± 39 flows/sec

TABLE II: Latency Breakdown

Component	Mean Latency ± Std. Dev.
Binding Query	2.41ms ± 0.97ms
Policy Query	2.52ms ± 0.85ms
Other PCP Processing	0.39ms ± 0.27ms
Proxy	0.16ms ± 0.72ms
Overall	5.73ms ± 3.39ms

In order to measure these metrics, we use the `cbench` synthetic OpenFlow controller benchmark [34], which we modified for compatibility with OpenFlow 1.3. The tool emulates an OpenFlow switch and sends packets with randomized headers to the control plane, with both latency and throughput measurement modes.

Table I summarizes our microbenchmarks: the flow-start latency is approximately 5.73ms (from `cbench` in latency mode) and DFI can handle approximately 1350 flows/sec (from `cbench` in throughput mode) before it is saturated. Note that the reported flow-start latency includes only the time for the flow to traverse DFI in one direction and does not include any additional time required by the actual SDN controller to route the flow. Additionally, this flow-start latency was measured when the system was otherwise idle. Table II shows the average time spent per flow during each of DFI's subtasks. This breakdown shows that most of the latency comes from queries to resolve binding information and determine applicable policy (about 2.5ms each). The other processing done by the PCP and DFI Proxy is insignificant (less than 0.6ms combined).

Time to First Byte. We now characterize the performance impact of using DFI in an SDN in terms of the latency imposed on the first packet of a flow (Time to First Byte, or TTFB). All packets after the first will be handled by flow rules in the switches, so this latency characterizes the primary impact of reactively installing SDN flow rules, as in DFI, on network traffic. The TTFB also effectively bounds the speed at which users can query network services and receive a response. We measure TTFB as a function of load on the network in order to characterize any degradation when the control plane becomes saturated. To do this, we perform a TCP connection from an end host and measure the time between sending the SYN and receiving the SYN-ACK; simultaneously, randomized Ethernet packets are sent into the data plane at varying rates as background traffic. Note that these TTFB measurements include the time for DFI and the SDN controller to process the flow in both directions.

Figure 4 depicts how TTFB varies as a function of the load

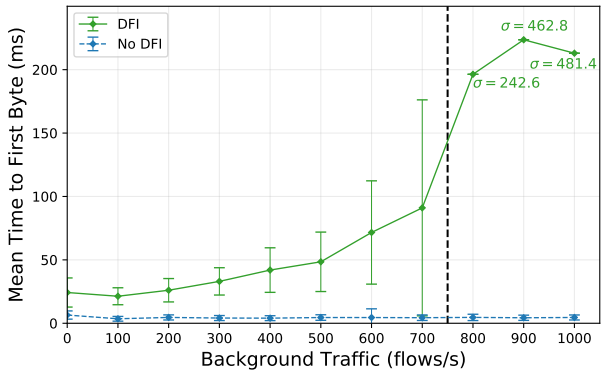


Fig. 4: Time to First Byte (TTFB) for new flows at different flow arrival rates. The dashed line indicates the point where DFI's queue begins to saturate and drop flows. Error bars show ± 1 standard deviation (σ) up until the saturation point, after which the standard deviation is high.

on the network, both with and without DFI in place. Without DFI, the TTFB is nearly constant at 4-6ms. While the SDN controller eventually becomes overloaded and queues packets, this occurs at significantly higher loads than we measure. With DFI, the TTFB starts at about 22ms and rises to about 85ms at 700 flows/sec. At higher rates, DFI begins to queue new flows waiting for binding or policy query responses, leading to the high variation observed above 800 flows/sec. The mean TTFB plateaus around 200ms because DFI has a limited queue size; flows arriving when the queue is full are dropped and must re-enter the DFI control plane upon retransmission. This saturation point suggests that DFI can support small enterprises since existing work [35], [36] has used 10 flows/sec/device (or 1000 flows/sec total) as a typical enterprise workload level. Scaling up could be achieved using multiple DFI Proxy and PCP instances.

B. Security Evaluation

We evaluate potential security benefits that a fine-grained, event-driven access control policy system like DFI can provide using a case study with self-propagating malware. We consider a scenario where the malware infects a foothold in a small enterprise network, and then tries to spread across the network over the course of a business day.

Threat Model. We consider a threat model where the SDN controller, switches, and core network services (DNS, DHCP, *etc.*) are secure and not compromised and end hosts are traditional enterprise desktops that are always on and connected to the network with users logging on and off throughout the day. These desktops may become infected by an automated worm that would attempt to infect and destroy as many machines as possible. This threat model is motivated by the recent resurgence of self-propagating malware, such as the NotPetya and WannaCry ransomware. While our discussion in the rest of this paper is focused on this threat model, we believe that other scenarios (wireless devices, BYOD devices) share the same fundamental issue: systems are overly privileged and dynamic signals exist that indicate when these privileges could be reduced.

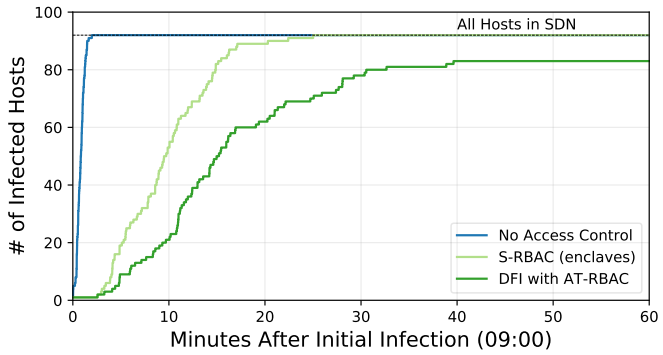
To simulate the threat of self-propogating malware, like NotPetya and WannaCry, we constructed a surrogate of the NotPetya malware (henceforth, the "worm") based on its propagation logic (see [37], [38]) to see how various access control policies reduce the spread of the infection.

At the start of the attack, we assume the worm has a foothold on one end host in the network. Once installed, it gathers a target list of end hosts and servers in the network through reconnaissance, and then tries to propagate to each target serially in a loop. The worm uses two vectors for propagation: exploitation of vulnerabilities on a target end host and credential theft. The exploit payload is sent first. If the exploit succeeds, the worm moves on to attacking the next target in the list. If it fails, the worm uses credentials cached on the local host to attempt to access the target remotely and install itself. A credential with "Local Administrator" privileges on the target must be cached on the source host for this to succeed. After looping through all targets, the worm waits three minutes before restarting. This proceeds over a duration of 10-60 minutes (randomly chosen) before the worm times out and stops propagating, as NotPetya does.

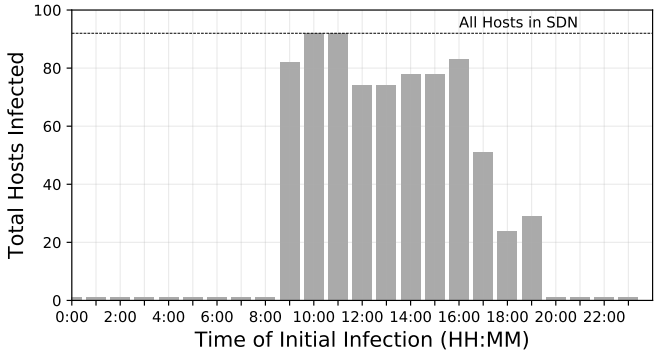
The goal of the threat is to spread to as many hosts as possible before the propagation times out, with no targets being more valuable than others. The target list is shuffled randomly on each infected host. We assume control-plane hosts are protected from reconnaissance by hosts in the SDN, and are therefore beyond the scope of this threat.

Network Testbed. In this study, our testbed is modeled after a small, operational enterprise network. It is built with VMware vSphere and includes 86 Windows 10 VMs acting as end hosts and 6 Windows server VMs supporting common enterprise services (*e.g.,* email, web proxy, file server). The data plane includes 14 OpenFlow switches implemented on CentOS 7 VMs with Open vSwitch 2.5.4, and 2 CentOS VMs running the OpenFlow control plane (ONOS 1.13 controller and DFI Proxy) and the DFI control-plane components. The network topology is a star, with a single core switch and 13 enclave switches internally connected to it. Nine of the enclaves support operational departments, with 9 hosts in each, while the remaining enclaves host servers and an smaller department with five hosts. One end host in each enclave (10/86 total) is configured to be vulnerable to the worm exploit, which falls within a typical range for patch compliance by organizations as previously reported by Symantec [39]. In addition, all servers are vulnerable in order to give them a vector for transmitting the worm, since they are otherwise defended against credential theft by configuration.

Users for the testbed end hosts are managed by an AD server in the data plane. Each end host has one unique, primary user, but other users in the same enclave (department) group have "Local Administrator" privileges on the host. Servers in the testbed have no primary users, and therefore no cached credentials. Log-on and log-off events for users on their



(a) Infections from a self-propagating malware under different network conditions.



(b) The impact of an infection using the AT-RBAC policy with DFI are conditioned on time.

Fig. 5: Results of DFI-enabled Policy on Testbed Infections

primary host are simulated over the course of the day, each being randomly assigned a unique time-series "script" that establishes when the user is logged on or off. These scripts were created by the authors based on a sample of their host interactions during the day, and form an anecdotal scenario for how the testbed network might be typically used. Each script contains at least two hours of being logged on during the first half of the work day (between 09:00-13:00). The randomized script assignment is reused between conditions.

Conditions. We evaluate a scenario where one end host in a departmental enclave becomes an infected foothold during the course of a business day. Each end host in the testbed simulated a unique authentication script that was randomly chosen and fixed between test iterations. We evaluate how the worm spreads when the foothold occurs at start of each hour in the day, under three policy conditions:

First, we consider a baseline condition of a fully-connected network with *no access control*. All traffic is allowed between hosts in the SDN.

Second, we consider DFI with *static, role-based network access control* (S-RBAC). In S-RBAC, access control is configured statically, indefinitely letting a host communicate with others within a logical enclave based on its role needs. In our implementation of S-RBAC, we install rules that allow incoming and outgoing flows for each host to: 1) all hosts in its own enclave, and 2) each of the servers for operational needs.

Finally, we consider a policy that is uniquely enabled by DFI called *authentication-triggered, role-based network access control* (AT-RBAC). In AT-RBAC, DFI enforces an access control policy that is specific to the user logged onto an end host. Role-based access for the user is allowed only after she authenticates and access is revoked upon logging off. When there is no user, flows are allowed only for a small set of services needed to authenticate (*i.e.,* DHCP, DNS, AD). We expect AT-RBAC to slow the worm when not all hosts have logged-on users, which is typical in realistic networks. In DFI's implementation of AT-RBAC, a sensor in the SDN detects authentication events from users on end hosts and sends them to the control plane. Seeing these events, a Policy Decision Point created for this policy sends or revokes commands (for log-on and log-off events, respectively) to/from the Policy Manager that allow the host incoming and outgoing flows for a role-based set, including: 1) all hosts in its own enclave, and 2) each of the servers.

Results and Discussion. In summary, the AT-RBAC policy uniquely enabled by DFI leads to fewer overall infections and a slower infection rate compared to the other policies. Given that the simulated user activity ensured morning activity on all hosts, the scenario probably demonstrates a conservative estimate of the benefit in a typical network. This slowdown could provide additional time for an incident response team to be notified and isolate infected hosts.

Results from the foothold starting at 09:00 represent how the policies slow the threat at the start of a work day. Figure 5a shows the first hour of the 09:00 infection for all test conditions. In the baseline condition with no access control, the first infection occurs after 1 second, and all end hosts and servers are infected after 2 minutes. In S-RBAC, the first infection does not occur until after 2.5 minutes. Initial attempts by the foothold to reach other hosts fail because the first targets exist in other logical enclaves, and are therefore denied by the role-based policy. The infection progresses after a server is infected and can transmit the worm to other enclaves, leading to full network infection after 25 minutes. In AT-RBAC, the first infection again takes 2.5 minutes due to the enclave RBAC, as in the S-RBAC condition. However, infecting other enclaves is slower than in S-RBAC: once a server is infected, it can only succeed at reaching a target host if that host has a logged-on user. As such, the worm's targets become "moving targets" whose reachability changes over time based on end-host usage. It takes the worm 40 minutes to infect 83 of the 92 hosts, with the worm propagation stopping (*i.e.,* ransomware "lock down") before it infects all hosts. After a post-hoc review, we determined that one enclave was not infected because its vulnerable host was not logged into until 10:46 – after all other infections had timed out.

Figure 5b illustrates how AT-RBAC, which is conditioned

on log-on and log-off events, provides a greater benefit when these events are sparse. In this scenario, the simulated log activity dwindles outside of usual business hours, and a foothold infected during this time cannot spread its infection before the worm times out. This is in strong contrast with S-RBAC or the baseline conditions, where infections initiated at any hour follow the same course as the 09:00 foothold demonstrated in Figure 5a, infecting of all hosts.

The scenario illustrates a benefit of access control approaches that make permissions dynamic over time, which is what DFI is designed to support. Both S-RBAC and AT-RBAC policies slow the rate of infection because it cannot spread directly from an end host into another enclave besides the servers. Yet, static policies like S-RBAC leave the network more vulnerable than is necessary during times when hosts can be effectively disconnected without impacting network operations, *e.g.,* outside work hours. These hours with no users represent the best case for AT-RBAC – flow rules are so restrictive that the foothold is isolated. In the worst case, AT-RBAC is equivalent to S-RBAC when all hosts have users logged on, but this is unlikely to last indefinitely in typical enterprise networks where users log on and off regularly.

VI. Related Work

As discussed in Section II, traditionally network access control has been highly static, focused on curated lists of firewall rules. Even approaches with a dynamic access-control check, like IEEE 802.1x [1] and products like Cisco Network Admission Control (NAC) [40] and Microsoft Network Access Protection (NAP) [41], conduct a single check before allowing coarse-grained access to the entire network. In contrast, DFI provides dynamic, per-flow access control via policies that are conditioned on network events, in place of static rules that may be overly permissive at times.

Software Defined Perimeter (SDP) technologies [12], [42]–[44] assume that most network assets are untrusted and thus users should authenticate to every server for which they need access. This is implemented at the application layer using application proxies; however, the lower-layer network remains static, with broad reachability. DFI differs by enforcing policy at Layer 2, dynamically limiting reachability at a lower level.

Unlike traditional networks, SDNs are designed to be centrally and programmatically controlled, providing an opportunity for improved access control techniques. Ethane [45] enforces access control at the per-flow level and enables policy specified with higher-level identifiers, as in DFI. However, the policies it uses to create flow rules are essentially static and do not adapt to events that might otherwise inform policy decisions. Similarly, `FLOWGUARD` [46] is a framework to detect and resolve conflicts in SDN firewall rules when the network state changes, but the intended flow policy is fundamentally static.

Precise Security Instrumentation (PSI) [47] leverages SDN to steer traffic to middleboxes providing varying levels of processing and inspection based on anomalous traffic features. PSI is capable of implementing a variety of expressive policies, much like DFI. However, it focuses on policies for traffic inspection and routing, while DFI provides the ability to dynamically adapt the network's access control to ongoing events. Additionally, PSI is controller-based, while DFI operates outside the controller to mitigate policy-bypass concerns.

PivotWall [48] combines SDN with information-flow control, enabling novel policies in the network. Using taint tracking, an end-host agent tags administrator-labeled resources as sensitive, and then alerts the SDN controller about flows initiated by tainted processes. The controller maintains a Network Information Flow Graph that is used to enforce information-flow control policies. These end-host events represent possible policy events for DFI PDPs, but DFI is aimed at a more general architecture and enables access control policy enforcement separated from, and with priority over, controller behavior.

Other efforts have investigated the security of SDN systems, or have leveraged the dynamism offered by SDNs to implement new defenses. Avant-Guard [49] provides access control mechanisms to disrupt control-plane saturation attacks. TopoGuard [23] and SPHINX [24] study attacks on the binding between an endpoint's MAC address and network location. SecureBinder [25] provides a solution to other attacks on identifier bindings by leveraging SDN's global view of the network. Programmable BYOD Security [50] uses the SDN for mobile-device access control. Security-Mode ONOS (SM-ONOS) [51] builds a permission system for SDN applications on top of ONOS. ConGuard [52] discovers Time of Check to Time of Use (TOCTTOU) bugs in SDN controllers. Finally, DELTA [53], NICE [54], and BEADS [55] provide frameworks for automated testing of SDN systems.

VII. Conclusion

Existing network access-control approaches are highly static and often coarse-grained. In this work, we have developed DFI, a system that supports event-driven, fine-grained dynamic access control policies using software-defined networking. DFI is implemented for OpenFlow networks, and properly handles consistency issues caused by frequent policy rule changes. At the same time, it provides high-level policy specification by resolving hostnames and usernames down to identifiers visible in the network traffic. Additionally, DFI's access control is independent of the SDN controller and does not require using a particular OpenFlow controller or trusting its integrity. We evaluated DFI's performance and show that DFI increases the time-to-first-byte latency for data transiting an SDN by 17.8ms under no load. This additional latency increases to 86.7ms at 700 flows/sec when saturation begins. This could scale to higher loads by running some control-plane components in parallel. Additionally, we experimentally evaluated a threat scenario using an authentication-triggered access control policy that is uniquely enabled by DFI, and found a decrease in both the infection rate and total infected machines from a NotPetya-like worm. These findings suggest that using DFI to enforce event-driven access control policies can provide improved network security over static approaches.

REFERENCES

[1] P. Congdon, B. Aboba, A. Smith, G. Zorn, and J. Roese, "IEEE 802.1 X remote authentication dial in user service (RADIUS) usage guidelines," 2003, RFC 3580.

[2] K. Sood and S. Hurley. (2017) NotPetya technical analysis – a triple threat: File encryption, MFT encryption, credential theft. CrowdStrike. [Online]. Available: https://www.crowdstrike.com/blog/petrwrap-ransomware-technical-analysis-triple-threat-file-encryption-mft/-encryption-credential-theft/

[3] A. Greenberg. (2018) The untold story of NotPetya, the most devastating cyberattack in history. Wired. [Online]. Available: https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/

[4] S. Hurley and K. Sood. (2017) NotPetya technical analysis part ii: Further findings and potential for MBR recovery. CrowdStrike. [Online]. Available: https://www.crowdstrike.com/blog/petrwrap-technical-analysis-part-2-further-findings-and-potential-for-mbr-recovery/

[5] Risk Based Security. (2018) Equifax breach: A wrap-up. [Online]. Available: https://www.riskbasedsecurity.com/2017/10/equifax-breach-a-wrap-up/

[6] Illusive Networks, "Attack Brief: Bangladesh Bank SWIFT Attack," Illusive Networks, Tech. Rep., 2016. [Online]. Available: http://cdn2.hubspot.net/hubfs/725085/Fact_Sheets/2016-09-ILL-1376--w-Attackerbrief-BangladeshSWIFT.pdf

[7] R. Altamini, N. Arora, and A. Kadi. (2015) Anthem Hack. Anthem. [Online]. Available: https://www.cs.bu.edu/~goldbe/teaching/HW55815/presos/anthem.pdf

[8] A. Jeng, "Minimizing damage from JP Morgan's data breach," SANS Institute, Tech. Rep., 2015.

[9] K. Jarvis and J. Milletary, "Inside a targeted point-of-sale data breach," Dell SecureWorks Counter Threat Unit, Tech. Rep., 2014.

[10] Trend Micro, "Countering the advanced persistent threat challenge with deep discovery," Trend Micro, Tech. Rep. 10, 2013.

[11] SophosLabs. (2018) SophosLabs 2018 malware forecast. Sophos. [Online]. Available: https://media.scmagazine.com/documents/321/sophos_2018_malware_forecast,__80124.pdf

[12] R. Ward and B. Beyer, "BeyondCorp: A new approach to enterprise security," *login*, vol. 39, pp. 5–11, 2014.

[13] "Cve-2009-1925," "Available from MITRE, CVE-ID CVE-2009-1925.", 2015. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2009-1925

[14] "Project Floodlight." [Online]. Available: http://www.projectfloodlight.org/floodlight/

[15] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "ONOS: towards an open, distributed SDN OS," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 1–6.

[16] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a model-driven SDN controller architecture," in *International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, June 2014, pp. 1–6.

[17] J. Amann and R. Sommer, "Providing dynamic control to passive network security monitoring," in *International Workshop on Recent Advances in Intrusion Detection (RAID)*. Springer, 2015, pp. 133–152.

[18] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. J. Clark, "Kinetic: Verifiable dynamic network control." in *NSDI*, 2015, pp. 59–72.

[19] Dell, Inc. (2015) Dell openflow deployment and user guide 3.0. [Online]. Available: http://topics-cdn.dell.com/pdf/force10-sw-defined-ntw_Deployment%20Guide3_en-us.pdf

[20] Shamus McGillicuddy. Pica8 doubles flow rule capacity in its new OpenFlow 1.3 switch. [Online]. Available: http://searchsdn.techtarget.com/news/2240214709/Pica8-doubles-flow-rule-capacity-in-its-new-OpenFlow-13-switch

[21] Centec Networks. (2017) Centec networks - SDN/OpenFlow switch - v330. [Online]. Available: http://www.centecnetworks.com/en/SolutionList.asp?ID=42

[22] Hewlett-Packard Development Company, L.P. (2015) HP switch software OpenFlow v1.3 administrator guide K/KA/WB 15.17. [Online]. Available: http://h20566.www2.hpe.com/hpsc/doc/public/display?sp4ts.oid=5354494&docLocale=en_US&docId=emr_na-c04656675

[23] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures." in *NDSS*, 2015.

[24] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting security attacks in software-defined networks," in *NDSS*, 2015.

[25] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, "Identifier binding attacks and defenses in software-defined networks," in *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 415–432.

[26] B. E. Ujcich, S. Jero, A. Edmundson, Q. Wang, R. Skowyra, J. Landry, A. Bates, W. H. Sanders, C. Nita-Rotaru, and H. Okhravi, "Cross-app poisoning in software-defined networking," in *Conference on Computer and Communications Security (CCS18)*, 2018, pp. 648–663.

[27] B. Yan, Y. Xu, and H. J. Chao, "Adaptive wildcard rule cache management for software-defined networks," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 962–975, April 2018.

[28] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *NSDI*. USENIX, 2013.

[29] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. M. Parulkar, "Can the production network be the testbed?" in *9th USENIX Symposium on Operating Systems Design and Implementation(OSDI)*, 2010, pp. 365–378.

[30] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *NSDI*, 2013, pp. 1–13.

[31] R. Durairajan, J. Sommers, and P. Barford, "Controller-Agnostic SDN Debugging," in *CoNEXT*, 2014.

[32] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-Defined Networks," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 87–101.

[33] G. Developers. (2018) Protocol buffers version 3 language specification. Google. [Online]. Available: https://developers.google.com/protocol-buffers/docs/reference/proto3-spec

[34] Mininet Project, "cbench," GitHub repository, 2013. [Online]. Available: https://github.com/mininet/oflops/tree/master/cbench

[35] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney, "A first look at modern enterprise traffic," in *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*. USENIX Association, 2005, pp. 2–2.

[36] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu, "Towards fine-grained network security forensics and diagnosis in the sdn era," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 3–16.

[37] Carbon Black research team. (2017) Technical analysis: Petya/NotPetya-ransomware. Carbon Black. [Online]. Available: https://www.carbonblack.com/2017/06/28/carbon-black-threat-research-technical-analysis-petya-notpetya-ransomware/

[38] J. Gajek. (2017) A closer look at Petya's/NotPetya's network spreading code. eSentire. [Online]. Available: https://www.esentire.com/blog/a-closer-look-at-petyasnotpetyas-network-spreading-code/

[39] Symantec. (2010) Patch management best practices. Symantec. [Online]. Available: https://support.symantec.com/en_US/article.HOWTO3124.html

[40] BigFix Client Compliance, "Cisco NAC," *BigFix, Inc., Apr*, vol. 25, 2005.

[41] (2005) Microsoft network access protection (NAP). Microsoft. [Online]. Available: http://www.microsoft.com/windowsserver2003/technologies/networking/nap/default.mspx

[42] "Software defined perimeter," Cloud Security Alliance, Tech. Rep., December 2013.

[43] W. Labs, "Software defined perimeter (SDP) implementation," 2017. [Online]. Available: http://www.waverleylabs.com/services/software-defined-perimeter/

[44] I. Vidder, "Software defined perimeter," 2017. [Online]. Available: https://www.vidder.com/software-defined-perimeter/

[45] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07. ACM, 2007, pp. 1–12.

[46] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FLOWGUARD: Building robust firewalls for software-defined networks," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. ACM, 2014, pp. 97–102.

[47] T. Yu, S. K. Fayaz, M. Collins, V. Sekar, and S. Seshan, "PSI: Precise security instrumentation for enterprise networks," in *Proc. NDSS*, 2017.
[48] T. OConnor, W. Enck, W. M. Petullo, and A. Verma, "Pivotwall: SDN-based information flow control," in *Proceedings of the Symposium on SDN Research*. ACM, 2018, p. 3.
[49] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the ACM CCS*, ser. CCS '13. ACM, 2013, pp. 413–424.
[50] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu, "Towards SDN-defined programmable BYOD (bring your own device) security," *NDSS'16*, 2016.
[51] C. Yoon, S. Shin, P. Porras, V. Yegneswaran, H. Kang, M. Fong, B. O'Connor, and T. Vachuska, "A security-mode for carrier-grade SDN controllers," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 461–473.
[52] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the brain: Races in the SDN control plane," in *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 451–468.
[53] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras, "DELTA: A security assessment framework for software-defined networks," in *Proceedings of NDSS*, vol. 17, 2017.
[54] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford *et al.*, "A NICE way to test openflow applications." in *NSDI*, vol. 12, no. 2012, 2012, pp. 127–140.
[55] S. Jero, X. Bu, C. Nita-Rotaru, H. Okhravi, R. Skowyra, and S. Fahmy, "BEADS: Automated attack discovery in OpenFlow-based SDN systems," in *Proc. of RAID'17*, 2017.