Exploring Host-based Software Defined Networking and its Applications

by

Douglas C. MacFarland

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

____

May 2015

APPROVED:

____

Professor Craig A. Shue, Major Thesis Advisor

____

Professor Krishna K. Venkatasubramanian, Thesis Reader

____

Professor Craig E. Wills, Head of Department

Abstract

Network operators need detailed understanding of their networks in order to ensure functionality and to mitigate security risks. Unfortunately, legacy networks are poorly suited to providing this understanding. While the software-defined networking paradigm has the potential to, existing switch-based implementations are unable to scale sufficiently to provide information in a fine-grained. Furthermore, as switches are inherently blind to the inner workings of hosts, significantly hindering an operator's ability to understand the true context behind network traffic.

In this work, we explore a host-based software-defined networking implementation. We evaluation our implementation, showing that it is able to scale beyond the capabilities of a switch-based implementation. Furthermore, we discuss various detailed network policies that network operators can write and enforce which are impossible in a switch-based implementation. We also implement and discuss an anti-reconnaissance system that can be deployed without any additional components.

Acknowledgements

Chiefly, I would like to thank Professor Craig Shue for his invaluable help, support, and, above all, patience as both a research advisor and a mentor during my time at WPI. Without him, I would not have been able to make it to where I am today. His guidance allowed me to discover the areas of Computer Science for which I am truly impassioned.

I would also like to thank my lab mates, without whom graduate school would have been a much less animate journey. Our discussions provided both crucial perspective and respite over the years, and I have always found our camaraderie to be a source of succor in otherwise trying times.

Finally, I would like to thank my parents, my sister, and my brother. Their unwavering encouragement and support throughout my life has always given me the the strength to push myself and to pursue my dreams.

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 author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

List of Tables

1 Introduction

Network operators need detailed understanding of their networks in order to ensure functionality and to mitigate security risks. The more information that operators have about the cause and context associated with network traffic, the more informed decisions they can make. Unfortunately, as most security systems in legacy networks are deployed at routers, not on each switch within a network. As a result, operators are blind to intra-subnetwork traffic, which solely traverses switches. This may allow an adversary to freely spread among hosts within a subnet without risk of detection. Even in the case that traffic is visible to the security systems, little to no host context is available, forcing decisions to be made using incomplete information and imperfect inferences gathered by inspecting transport layer port information and/or deep packet inspection.

The software-defined networking (SDN) paradigm gives network operators complete control over their network traffic. The OpenFlow protocol [37], the most popular SDN implementation, allows a commodity computer to make the control-plane decisions for OpenFlow-compatible switches. When such a switch does not know how to forward a packet, it passes the packet to the OpenFlow controller for instruction. This "elevation" process can be utilized to inform the controller of all flows within the network, as well as providing a centralized location from which to perform access control. However, as OpenFlow is a switch-based implementation, it cannot grant the controller any visibility into the end-hosts.

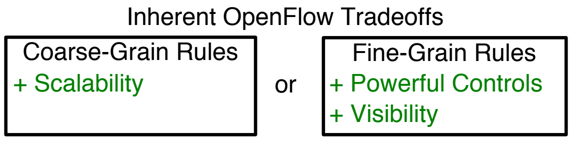

Figure 1: OpenFlow's infrastructure-focus creates inherent tradeoffs between scalability and fine-grain controls.

Furthermore, OpenFlow has difficult scaling sufficiently to provide fine-grained rules (which include the transport layer protocol, source and destination IP addresses, and source and destination ports) in larger networks. An OpenFlow switch is an aggregation point for multiple hosts worth of traffic and must maintain rules for all of the traffic that can pass through it. Thus, while the OpenFlow protocol does allow for rules that match on fine-grain network flows, the use of such rules is discouraged. Researchers instead suggest using coarse-grained rules using wildcards in some of the fields to mitigate this scalability concern [17]. As an example, even a high-end data center switch, such as the IBM RackSwitch G8264 switch [29] which can store a maximum of 97,000 OpenFlow rules [23], would be challenge to meet peak demand in large enterprise networks, which may handle over 100 million flows in a day [25]. Using fine-grained flows in OpenFlow can result in a loss of security, as adversaries can cause a denial-of-service attack by establishing many long-lived flows that force switches into a thrashing state [46].

Even with a scalable fine-grained SDN solution, it is difficult to stop an adversary that has compromised a host from accessing networking resources that that host is authorized for. Distinguishing between legitimate and malicious applications is beyond the scope of this work. Instead, we explore applying our SDN implementation to creating an anti-reconnaissance application that leverages the knowledge of legitimate users.

In this work, explore creating a host-based SDN implementation, capable of providing detailed host context and scaling to large networks. By moving the SDN functionality from network switches and routers into hosts, illustrated in Figure 2, we leverage the state already maintained by the host as a part of normal network communication. Ultimately, by merging the SDN functionality into the end-hosts themselves, we make the following contributions:

- **Host context for enhanced control decisions:** Our approach will enable network operators to create nuanced policies for functions such as access control. For flows internal to the organization, network operators will be able to write policies specifying the necessary host context on both the source and destination hosts. For flows where one endpoint is external to the organization, network operators will still be able to gather host context for the internal endpoint and will retain full control of the flow while it remains inside the organization's network. Our approach is extensible, allowing for arbitrary modules to be run on hosts to provide any desired context. Future work may include providing highly detailed causal information, such as "USER clicked a button, with button text 'ACTION,' in APPLICATION to initiate this network flow." Such policies may be specified using security language, such as Flow-based Security Language (FSL) [26], for formal analysis.

- **Scalable, fine-grained flow-based rules:** By incorporating the SDN functionality into end-hosts, we remove the need to maintain per-flow SDN state on switches and routers. The organization's switches and routers can function as they do today in a legacy network. Furthermore, we only add minimal new state on the end-hosts, as they already maintain per-flow state in the kernel for the purpose of connection tracking. Accordingly, we can utilize fine-grain flow rules for purposes such as access control without scalability concerns.

- **Inexpensive and immediate deployment options for enterprise-sized networks:** By shifting the implementation from network infrastructure to software on end-hosts, we enable organizations to quickly, inexpensively, and incrementally, deploy the approach. Standard enterprise software deployment tools can be used to install an SDN agent on the organization's systems, granting immediate control over the systems' networking while avoiding additional capital costs [28].

- **Application for anti-reconnaissance:** Our SDN implementation enables us to create a network moving target defense system capable of hindering an adversary's ability to spread within the organizational network without impacting legitimate users. The system is implemented as an application running on our SDN controller, requiring no further modifications to hosts.

2 Background and Related Work

We group the background and related work into five categories: containment, monitoring, dynamic access control, gathering host context, and anti-reconnaissance. We now describe each.

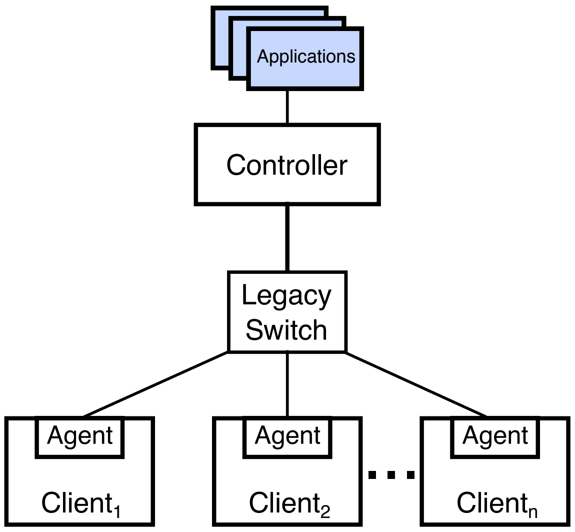

Figure 2: Integrating SDN functionality in host-based agents allows use of legacy switch infrastructure.

2.1 Network Containment

Virtual local area networks (VLANs) are frequently used to separate trusted and untrusted hosts by placing them in separate broadcast domains [15]. Unfortunately, they require manual configuration and can introduce complex routing inefficiencies [21]. Another proposed technique is microsubnetting, where each host placed in a separate /30 subnetwork, which forces all inter-host communication to go through the router [8]. Once hosts have been approved by an authentication and vulnerability check, it can be relocated to a normal subnetwork. Other work on sensor networks placed hosts with a high volume of traffic between each other in the same subnetwork to leverage the performance benefits of the increased locality [24].

However, while these techniques can isolate hosts into groups, they are still ineffective at monitoring the traffic within the group. Placing each host in its own group is also generally not a feasible solution. For example, placing each host in its own VLAN forces all traffic through a router, which can become overburdened and erases the benefits of being able to communicate over fully-switched infrastructure. Furthermore, they enforce a host level access control policy, instead of the finer-grained flow level. While our approach will allow host-based access control, we also enable decisions to be made using flow level knowledge.

OpenFlow, a software-defined networking (SDN) implementation, provides a mechanism to separate the control and data planes in networking infrastructure. Switches and routers supporting OpenFlow maintain a cache of rules used to make a forwarding decision [37]. Whenever a packet arrives for which no rule in the cache matches, it is "elevated" to the OpenFlow controller, which makes a routing decision and adds the appropriate rules to the cache. Alternatively, the controller can choose to discard the packet. This elevation property can be used to gain understanding of host connectivity [40], which can be leveraged to gain additional situational understanding.

While powerful, using fine-grained matching introduces considerable scalability concerns. An

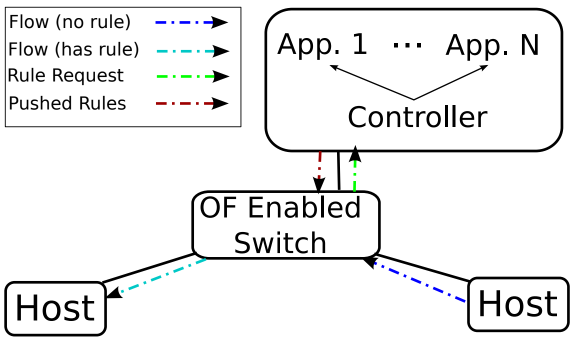

Figure 3: Software-defined networking enables a logically centralized viewpoint of the network.

extensive study on the scalability of OpenFlow by Curtis *et al.* [17] found that commodity OpenFlow switches were unable to support enough fine-grained flow rules to handle the daily traffic of organizations. To account for this, the authors advocated the use of wildcards to create broad rules when possible. While this is acceptable from a pure performance point of view, it results in a loss of visibility into and control over network traffic.

As an example of the limited flow state available in OpenFlow hardware, a recent work by Kuzniar *et al.*described three OpenFlow switches that had hardware support for between 750 and 2,000 flow rules and software support for an unspecified number of rules. Even top-end OpenFlow switches, such as the IBM RackSwitch G8264 switch [29], have a maximum of 97,000 OpenFlow rules [23]. Furthermore, the limited hardware flow capacity can make OpenFlow switches vulnerable to denial of service attacks [46].

Our implementation allows for the adaptability of software-defined networking, without the scalability concerns of switch-based implementations, such as OpenFlow.

2.2 Network Monitoring

Network intrusion detection systems (NIDS) can monitor all traffic that transits them, and are thus commonly installed on routers or switch span ports. To gain full visibility of the traffic within its network, an organization could place a NIDS on every switch in their network. However, this would be both expensive and hard to coordinate due to the potentially large number of switches. Furthermore, each NIDS would see only the traffic on that switch, and there would a great deal of duplicate traffic reported the NIDS on each switch the traffic traverses. This makes this an unappealing approach for organizations. However, the existing placement leaves an organization unable to monitor intra-subnetwork traffic, allowing for an adversary to covertly spread within that subnetwork [51].

Our solution will allow flows to be directed through network way points, such as a NIDS, in a manner comparable to OpenFlow. As noted in the previous section, we can do so even for fine-grained flows.

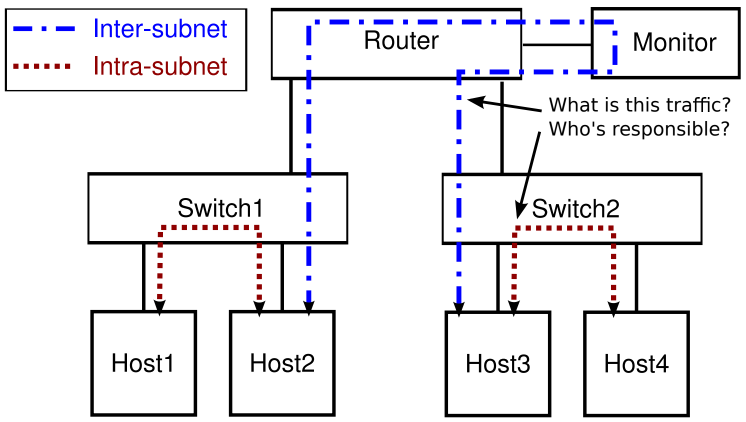

Figure 4: Network monitors are only able to monitor a subset of the traffic within the network.

2.3 Dynamic Access Control

Existing access control policies are generally statically configured and do not consider previous network history. However, dissertation, using proper word processing techniques. Consult the helpful hints available on the ETD Web site, a an organization can use information about its network to influence a more secure dynamic access control policy.

One work explores detecting scanning malware by monitoring for network connections not preceded by a DNS query [54]. Such connections were considered anomalous and triggered an alert. Effort was made to try and associate connections with DNS queries and to reduce the false-positive rate. However, the system was solely a detection mechanism.

Another work utilized a DNS query as a capability token [50]. Connections to a host for which no DNS query was issued are disallowed or sent to a honeypot, blocking cases such as IP-based scanning malware. An intermediate NAT device was utilized perform the appropriate translation.

Prior work, such as Resonance [42], has explored access control systems leveraging OpenFlow. Monitoring takes place on the network infrastructure itself, and allows flow-level information to be collected and sent to a central location. The programmable switches also allow flow-level access control decisions to be made prior to the traffic even reaching the host.

We will use host-based agents coordinated by a central controller to provide many of the benefits of these works, without requiring alterations of the network infrastructure and with less risk of the central point becoming a bottleneck.

2.4 Extracting Context from End-Hosts

Ethane [14] is an early SDN implementation designed to enhance network security by allowing network operators to write security policies including details such as users, end-hosts, and access points. Unfortunately, as a switch-based approach, Ethane doesn't have the ability to gather the necessary information from the end-hosts for the policy. Our work embraces the same ideals as Ethane while instrumenting the end-hosts to enable the enforcement of policies specifying host context.

HoNe [20] provides process attribution by linking network traffic to processes. However, HoNe is not centrally coordinated, does not support arbitrary host context, and does not enable SDN functionality.

The `ident` protocol [31] allows remote systems to query a host to learn what user on the system initiated a given application, which was particularly useful for multi-user systems. Naous *et al.* [41] proposed an extension of the `ident` 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 in an OpenFlow-enabled network in which queries for host-context were done by the controller when it receives a rule request from a switch. Our work also has the goal of fusing host-context with network traffic. However, `ident++` neither describes nor evaluates an implementation of its approach, nor does it discuss how it overcomes the scalability concerns for fine-grained flows in a switch-based SDN implementation. Our approach is host-based, allowing context to be provided proactively instead of reactively and addresses scalability by reducing the amount of state each entity needs to keep track of. We also evaluate our implementation, both via a native OS solution and a bump-in-the-wire implementation.

Other works have focused on the context available on end-hosts and how to extract this information to better understand users. Examples range from collecting mouse-clicks and keyboard presses [16] to application-specific implementations such as the user's interaction with a web browser [36,55], with more advanced approaches digging deeper into systems to gather information such as window titles or the location of certain GUI elements like an accept button [11,49].

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.

2.5 Anti-Reconnaissance

Reconnaissance is the first phase in any network intrusion [9]. By hindering an adversary's ability to gather information on other targets, we can effectively hinder the adversary's efforts to spread to other hosts in the network.

One technique is network address space randomization (NASR). Similar to the address space layout randomization (ASLR) technique used to protect against buffer overflow attacks [47], NASR works by changing the address of a machine in some randomized fashion. This limits or removed an adversary's ability to build up knowledge of IP address over time, and has been shown to negatively impact scanning malware [4]. NASR can be implemented in several different ways, depending on the kind of randomization desired.

Prior work, focused on defeating hit-list scanning malware, utilized DHCP to change the IP address of the host over time [7]. However, this change can disrupt existing connections. To handle this, placing an intermediate NAT-like box in the network to transparently transition to the new address over time was suggested [7]. The NAT-like device provided similar address translating behavior, with additional logic to preserve old addresses while they are still in use by previously established connections.

Other work utilized DNS and an intermediate NAT device [50]. Rather than changing out the DHCP lease to achieve randomization, the IP address contained in the DNS reply for the host is rotated randomly. The NAT device is notified and will translate requests going to that IP address to the host's fixed internal address. This has the benefit of not requiring an address to be monitored to see if there are remaining connections before expiring the address and utilizes existing technology.

Additional works leverage software-defined networking. Similar to the work by Shue *et al.* [50], the work by Al *et al.* [6] and Jafarian *et al.* [30] populates DNS records with random virtual IP addresses, which are then used for routing and are cryptographically tied to the host's real IP address. The use of OpenFlow allows routing to properly take place without an intermediate translation, but incurs the previously discussed limitations.

We perform the randomization at the hosts, eliminating the need for an intermediate device which can form a choke point in the network and allowing deployment on existing network infrastructure. As each host only needs to retain information about its own connections, this also allows us to scale much better than an intermediate device, which will have to retain state for many, if not all, hosts in the network. This has the added benefit of disallowing an adversary from gaining information about whom the host is talking to. Furthermore, since MAC addresses can be used to identify hosts, even if IP addresses are randomized, we also enable the randomization MAC addresses.

3 Threat Model: User-Level Adversary

Our threat model is designed to be useful in common organizational scenarios. We discuss ways to relax some of the stronger assumptions of our model, though we do not explore them in this work. Our model primarily considers an external adversary that has compromised a user-level account on a host within the organization and who is interested in spreading within the network. We make the following two key assumptions:

- **Trusted Operating System:** Our strongest assumption is that no administrative-level compromises occur, reflecting the best practice of "least user privilege" [45] under which malware will only be able to run with the compromised user's privileges. This is a common assumption among host-based defense systems, such as anti-virus, firewall, and host-based intrusion detection software. We can relax this assumption using techniques such as virtualization or bump-in-the-wire solutions that prevent an adversary with administrative access to a host from compromising our software. However, even in the event that an adversary achieves a root compromise, the controller can detect if an access control and/or routing policy is not adhered to as long as all the involved hosts are not also administratively compromised.

- **No Physical Insider Threats:** While we consider an adversary that has compromised a user account within the organization, and this adversary may attempt to masquerade as that user, we assume that the adversary is inherently an outsider and does not have a priori knowledge of internally restricted information. Furthermore, our modeled adversary does not have physical access within the organization. Such an adversary could physically modify systems and hardware to bypass our implementation. While such an adversary may be considered in the future, we note that many attacks are remotely launched.

While we focus on organizationally managed devices, for which least user privileges are enforced, we can support both legacy and personally managed devices by placing such devices in individual VLANs and proxying network traffic through a middlebox running our software. While this approach may limit performance, it enables all of our flow-management capabilities, while allowing an organization to incrementally deploy our approach.

4 Approach

Our host-based approach moves all of the fine-grain rule matching and control to system-level software agents running on each of the hosts. Since end-hosts must already manage per-flow state to manage the connection, as in TCP connections, the approach has only minimal impact on end-host scalability and processing. In particular, Linux stores connection state and associates packets with flows using `conntrack`, which we can programatically interface with.

4.1 Core SDN Functionality

Our host agent does not require special kernel or application modifications. However, the agent does run with privileged user execution, allowing it to manage the system's configuration and operation. Similar agent-based system administration tools are popular in large enterprises [39] 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.

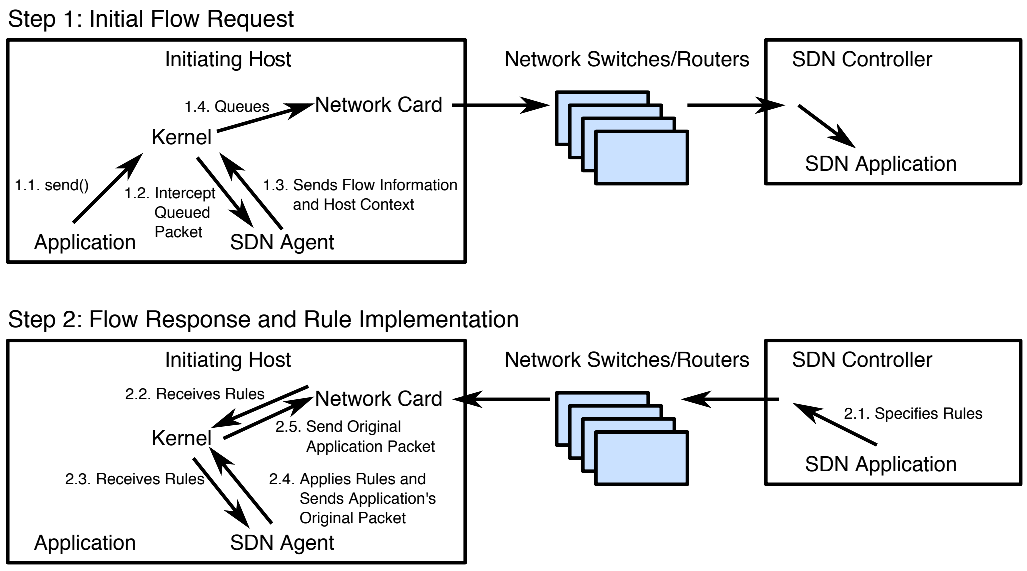

Figure 5: Overview of agent communication and control model.

To be an SDN implementation, our host agent needs to be able to programatically route packets based on rules assigned by a logically centralized controller. In Figure 5, we provide an overview of how the SDN agent manages an end-host. The process begins when a user application on the initiating host attempts to create a new connection. The initial packet in the connection will not match any existing kernel flows, represented as a tuple of the form `(source IP address, destination IP address, transport protocol, source port, destination port)`. Accordingly, while the packet is queued for transmission in the OS kernel, our SDN agent will extract the packet from the kernel queue[1].

Once the agent has intercepted the application's packet, it analyzes it and determines the context for the communication. The agent is extensible and can encode and transmit arbitrary host context from any data source on the end-host. As an initial instantiation of the approach, we create an agent that extracts the associated process, including the path to the process's executable, and the information about the user and group of the process owner. The agent then transmits a message to the SDN controller, which contains the flow tuple and the extracted host context, and requests instructions from the controller. If desired, the agent could include the packet payload as well. The agent queues the packet locally until it receives a response from the controller.

Upon receiving a rule request from a host agent, the SDN controller shares the request with any applications subscribed to the message type, much like in the OpenFlow architecture. Once the controller receives feedback from the applications, it returns one or more rules to the host agent describing the appropriate action that the host agent should take for subsequent packets in that flow. The controller may specify a rule timeout, after which the host agent will expire the cached rule. This will cause the host agent to again query the SDN controller for instructions, since the agent will have no rules corresponding to the traffic. If the flow naturally expires in the host agent before the controller's timeout, the host will send notification to the controller to ensure the controller has a correct view of the network. This approach is superior to OpenFlow in which the controller does not get explicit feedback when the hosts terminate their connections.

Once the host agent receives a response from the SDN controller, it applies the settings by updating the flow state, NAT or firewall rules, or even routing tables as necessary to implement the rules specified by the controller. Once this state is in place, the agent removes its interception markings for the flow and re-queues the original packet from the application, allowing the packet to be transmitted in accordance with the new policy implemented in the host.

In our system, host agents react to new connections in the same manner, regardless of whether the host is acting as the initiator or responder in the connection. As a result, the controller gets insight into all deploying hosts, even if they are communicating to external hosts that do not implement the protocol. For communication between two deploying hosts in the network, the controller gets insight from both the initiating host and the responding host, granting it two opportunities to review the connection. In the case of two deploying systems, the controller has full flow details when it receives the responder's message, since it can fuse the context with the initiator's earlier message. In our initial approach, this allows the controller to know the user and application that instigated the request on the initiator host and the user and application on the

[1]This packet interception approach is available to privileged applications on multiple operating systems (e.g., using `divert` sockets in BSD systems or using the `netfilter_queue` library in Linux system).

responder host that will service the request. The controller can then authorize the flow or send a rule to discard the message at the responder before it can reach the application. This allows for policies to incorporate details about the service being connected to, not just about the connecting application. For example, an administrator may want a particular web server to only be accessible by administratively-installed web browsers, prohibiting the use of a command-line tool such as `wget`, without disallowing the use of such a tool for any other servers. As a further optimization when a flow is denied, the controller can update the rules on the initiator to ensure an unauthorized flow never leaves the initiating host.

The system can also be utilized if only one of two communicating hosts is running the agent. While the host context will only available for the host running the agent, powerful policies can still be deployed. For example, an organization could prevent hosts from within its network from connecting to external web servers with outdated or vulnerable versions of web browsers.

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 [1]. 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 [32, 52].

While our approach has largely focused on the network functionality of the agents, the retrieval of host context plays an important role. The host agent can be customized to gather context based on the type of operating system in use on the host. The agent may perform arbitrary pre-programmed commands to obtain information from system data structures (such as examining information in `/proc` on a Linux OS) or query any databases created by software on the host system. As a result, the agent can transmit any information gather by sensors on the host as long as those sensors provide an API for the agent to query. This allows organizations to customize what context will be provided to their controllers so that they can write policy on the controller to meet their needs.

4.2 Host Context

As our host agent can run directly on a host, it is ideally placed to gather context from the host before elevating packets to the controller. Using this insight, we enable controllers to make decisions based on information that a switched-based implementation fundamentally cannot, such as the user or application responsible for the packet. When the agent receives a packet that is going to be elevated to the controller, it can first send the packet to a set of plugins running on the host.

As illustrated in Figure 6, plugins can be dynamically registered with the host agent. Each plugin will receive a copy of the packet and can return arbitrary information to be passed to the controller. Different hosts can run different plugins and each application running on the controller

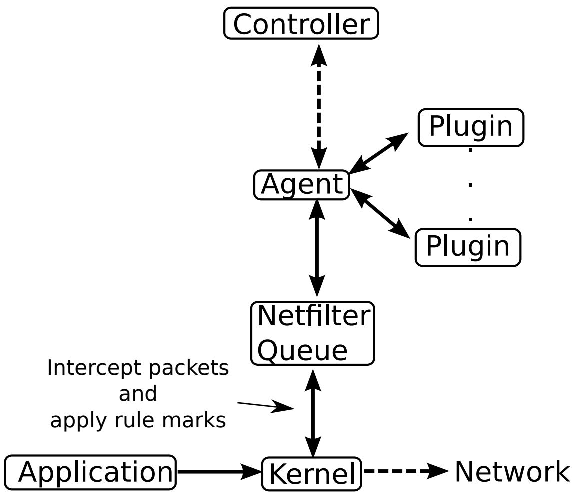

Figure 6: Plugins can be dynamically registered with the agent.

is free to only consider the context relevant to its decision.

4.3 Anti-Reconnaissance

Network reconnaissance is the initial goal of an adversary [9] once they have established a foothold within the organization and have bypassed the external defenses. The knowledge they gain during that phase allows them to use their foothold pivot and compromise more hosts. If we hinder the adversary from performing reconnaissance, specifically of network addressing information such as MAC and IP addresses, then we can hinder the adversary's attempts to exploit other hosts from their internal foothold. Furthermore, by hindering the adversary in the very first stage of the cyber kill chain, we can help minimize the cost to contain and/or recover from the breach.

While helpful, access control policies are often not enough, as the adversary inherently has all the privileges of the compromised account. To solve this, we need to leverage information that only a valid user would know and hinder an adversary's attempts to gaining that knowledge, even if they are able to monitor the system of a valid user. We focus on hindering an adversary's ability to discover usable network addressing information, which are necessary for them to establishing connections of their own.

We consider two possible viewpoints the adversary can have in the network. In the first, the adversary has managed to gain user-level access to a legitimate host, and is passively monitoring the connections to and from the host. This can be done without root permissions using a tool such as `netstat`. In the second, the adversary can view traffic between other hosts, allowing it to see all the addressing information in those packets. Examples include cases where an adversary is a part of the network, without being one of the organization's hosts.

Our approach operates by utilizing synthetic addressing information in place of the real addressing information. For simplicity, the synthetic information can be considered to be a randomly chosen valid value. This synthetic information will be the only addressing information that the adversary can observe and will not be valid for use outside of the connection it was observed in, vastly reducing the information's utility to the adversary. We discuss how to actually select the synthetic values in Section 6.6.2. Our discussion here relies on hosts issuing a DNS response prior to initiating a connection. If the DNS response is not modified, an adversary monitoring a legitimate host can learn the real IP addresses of servers.

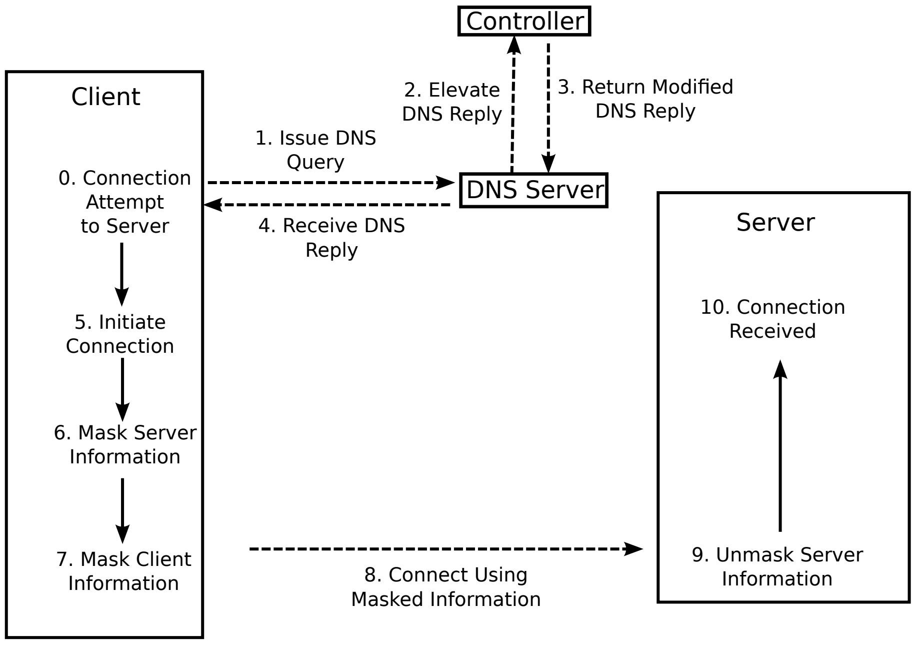
Controller
2. Elevate DNS Reply
3. Return Modified DNS Reply
Client
1. Issue DNS Query
0. Connection Attempt to Server
DNS Server
4. Receive DNS Reply
Server
10. Connection Received
5. Initiate Connection
6. Mask Server Information
7. Mask Client Information
8. Connect Using Masked Information
9. Unmask Server Information

Figure 7: High-level overview of the anti-reconnaissance process. For simplicity, the elevation of the connection to the controller has been omitted.

When the organization's DNS server issues an answer to a DNS request, it elevates that answer to the controller. The anti-reconnaissance application running on the controller substitutes the real IP address of the server with a synthetic one, and issue commands to both the client and server, informing them of the synthetic IP/MAC addresses to be used for the connection. The modified DNS response is then send back to the DNS server, which forwards it on to the client.

When the host receives the DNS response, it attempts to establish a connection to that address. As the address is synthetic, an adversary that has compromised the host will not be able to learn the real IP address. The corresponding server will have been configured to respond to that IP via ARP, despite the IP address being a synthetic one.

As the packets being sent to the server transit the client's kernel, the source and destination IP and MAC addresses are mapped to the new synthetic values chosen by the controller. As an adversary that has compromised the host will only have user-level privileges, it will be unaware of the mappings taking place.

Mapping the source IP and MAC addresses will hinder an adversary monitoring the network from identifying the client, while mapping the destination addresses hinders that adversary from identifying the server. Furthermore, since the already synthetic (via the DNS modification) destination address is mapped to another synthetic address, an adversary that is both monitoring the network and has compromised the client will have difficulty associating the traffic with the result of the DNS lookup, as different IP addresses will be used.

Since observing the synthetic addresses will not enable an adversary to connect to other hosts, it is forced to engage the DNS server and will need to know valid DNS host names. Due to this, DNS host names will serve as an access token. If the DNS host names are chosen such that they are difficult for an outsider to guess, an adversary must spend more effort to try and gather them from a legitimate user. Even then, the adversary is limited to discovering other hosts that the legitimate user accesses.

5 Implementation

We created an initial implementation of our approach on the Ubuntu Linux operating system using the Python scripting language for the host agents and controller. To enable communication between each system, we used asynchronous messaging with the Twisted framework [53]. These components were not optimized for performance and thus are conservative estimates of what would be possible in a production implementation.

5.1 Host Agent

The host agents are the key component in our architecture. The agents run on each end-host with root privileges. Once the agent starts, it waits for the host to be assigned an IP address by monitoring the host's network interfaces. The agent then establishes an SSL-protected TCP connection with the controller and allows the controller to proactively transmit any network configuration instructions. The host agent then begins to intercept all network communication that does not match the pre-defined rules provided by the controller.

The agent distinguishes between flows it has and has not processed via connection marks. Initially, a connection's mark is unset. Once the agent has installed a rule that matches the connection, it sets that connection's mark, which is shared between both the incoming and outgoing flows of that connection and is responsible for the routing decisions for those flows.

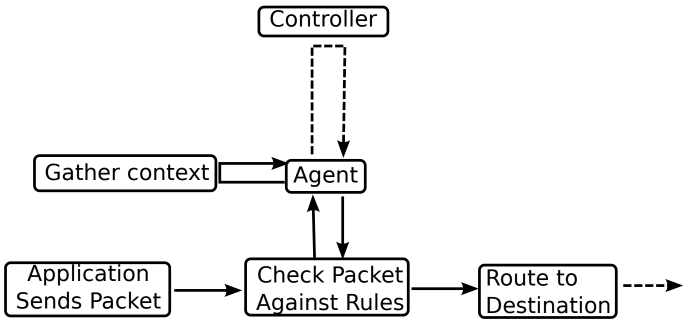

Figure 8: High-level overview of the host agent process

5.2 Elevation

The host agent will intercept any packet originating from, passing through, or destined to the host unless the network flow has an existing rule that matches the flow. To ensure this, we leverage `conntrack`, `iptables`, and `netfilter_queue`.

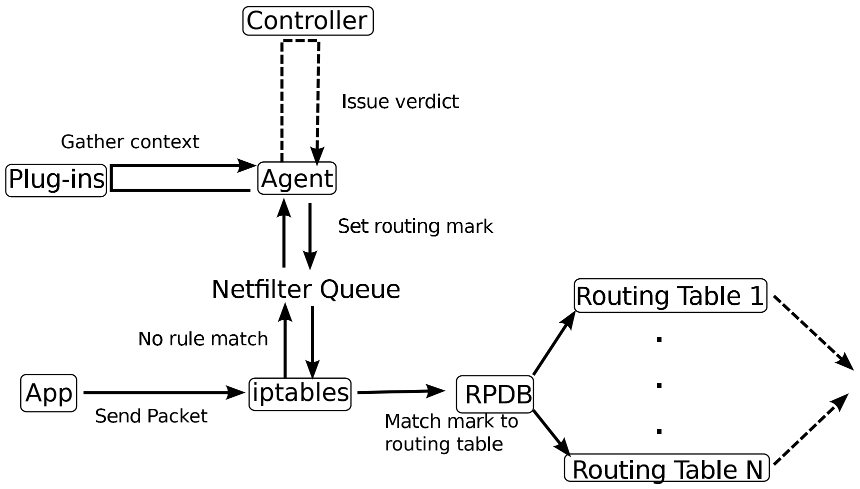

Figure 9: Overview of the host agent process

Packets are grouped into network flows by `conntrack`, which also keeps track of that flow's connection mark. A connection mark is a 32-bit integer that is automatically set for all packets in that connection prior to the packet being processed by `iptables`. We use the connection mark to identify the appropriate forwarding rule, with a mark of zero (the default value for new connections) indicating that no rule is set. The host agent must keep track of any preemptively inserted rules, as the matching connections will not yet exist and thus have no connection mark to set.

When a packet is seen that does not match any existing flows, `conntrack` establishes state for the new connection in the kernel. While logically, flows are a 5-tuple of `(transport protocol, source IP, destination IP, source port, destination port)`, `conntrack` identifies connections by a 9-tuple in which the `source IP, destination IP, source port,` and `destination port` are tracked separately for incoming and outgoing packets. This takes into account the fact that all NAT translations happen after `conntrack` sees the packet and thus incoming and outgoing packets would otherwise appear to have different addresses and would not be properly identified as belonging to the same connection. This also means that NAT translations cannot be altered for an existing connection in a non-disruptive fashion, as they are an inherent part of that connection's identifying state.

Using `iptables`, we can check a packet for a connection mark. If it has a non-zero connection mark, we know it has a matching rule and we set the forwarding mark, which is also a 32-bit integer, to be equal to the connection mark. We discuss how the forwarding mark is used in Section 5.3. However, if the connection mark is zero, we pass the packet off to `netfilter_queue`.

The `netfilter_queue` library allows us to remove from the host's network stack and adds it to a queue. The queues provide a `netlink` socket, a common Linux kernel interface for inter-process

```
iptables -A OUTPUT -t mangle -m connmark --mark 0 -j NFQUEUE
iptables -A OUTPUT -t mangle -o eth0 -m connmark --mark DENIED -j DROP
iptables -A OUTPUT -t mangle -o eth0 -j CONNMARK --restore-mark
```

Figure 10: The firewall rules necessary to intercept outgoing packets in unmarked connections, where DENIED is the integer mark for blocked connections

communication, that a administratively executed user space application can listen on. When a packet arrives, that application can remove that packet, perform arbitrary processing and/or modifications to the packet, and then resubmit the packet to the kernel with various verdicts such as accept and drop. If the verdict is accept, the packet continues to be processed in the network stack. If the verdict is drop, the packet is discarded. The packets in the queue have a unique identifier, allowing the receive/process/resubmit process to be done asynchronously. Multiple packets can be received and simultaneously processed before any of them are resubmitted and packets do not need to be resubmitted to the kernel in the order that they were received.

We configure our host agent to listen on the queue. When a packet is received, the host agent performs some set of behavior defined by the configured plugins (see Section 5.7 for more details on plugins) before passing the packet and associated information on to the controller. Since the connection mark remains zero until the controller issues a verdict, packets in that connection that arrive prior to that time will also be enqueued and elevated.

When the host agent receives the forwarding rule from the controller, it installs them, sets the appropriate connection mark using `conntrack`, and then resubmits the packet to the kernel. Note that while the connection mark uniquely identifies a rule for a flow, the same mark value can be shared by many connections going to the same next hop.

When the fine-grain rule specific to the flow expires, the host agent will remove the connection's mark, which will cause the next packet in the flow to be intercepted just like it is a new flow. This will re-initiate the original elevation process, requiring repeated approval for long-running flows if desired.

5.3 Routing and Forwarding

When approving a network flow, the controller may specify the routing behavior of the flow. The controller can specify that the host should use its default routing table to transmit the packet: this will forward packets directly to other hosts (for intra-subnet traffic) or relay via the host's default gateway (for traffic outside the subnet).

The controller also has the option of specifying an arbitrary next hop for the flow, bypassing the default routing table. We first create a unique routing table for each next hop, specifying the next hop as the routing table's gateway. We then use the Linux `ip-rule` command to manage the routing policy database (RPDB). The routing policy rules specified by `ip-rule` can utilize various networking information, as well as the forwarding mark set by `iptables`. When used in conjunction, `ip-rule` and `iptables` allow the specification of incredibly rich policy rules.

We create a policy rule indicating that the connection marking from the controller should be re-used to determine which routing table to use, allowing the controller to specify the default

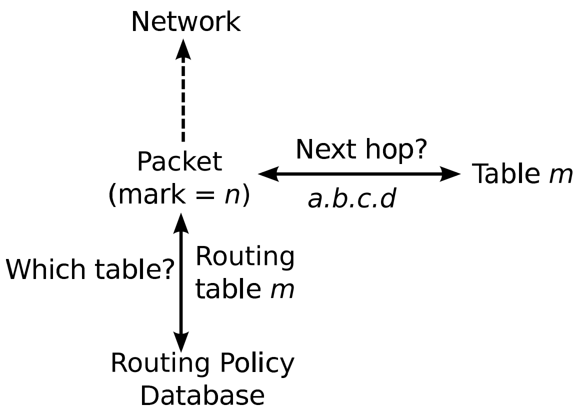

Figure 11: Routing decisions are based off the the connection mark

route 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 (such as an IDS, proxy, or firewall). By changing the connection mark for a flow, which can be done without interrupting the flow, the controller may dynamically alter the forwarding instructions for a flow at will.

The state required to specify a next hop includes 1) a routing policy rule, 2) a corresponding routing table, and 3) the firewall rule to set packets forwarding mark to be the connection mark. As we only need a single firewall rule to assign the forwarding marks, and the policy rule and routing table can be shared among connections, we have no need for additional per-flow state beyond what is natively used in the kernel. The association of packets into flows is already handled by `conntrack` and only a fixed number of rules are required to intercept packets via `netfilter_queue`. Furthermore, the userspace components (host agent, `netfilter_queue`, context plugins) are only consulted when there is no matching rule for a flow, in which case the flow's connection mark will be zero. Once there is a matching rule, all processing takes place in the kernel.

5.4 Rule Timeouts

As in the OpenFlow protocol, our SDN controller may specify flow rules with both a soft timeout (a timeout for idle flows) and a hard timeout (which triggers regardless of flow activity).

In our implementation, a soft-timeout occurs when the connection naturally expires in the kernel. When this happens, a `CONNTRACK_DESTROY` event is generated, which we can listen to using the `netfilter-conntrack` library. When we receive such an event, the agent informs the controller.

By default, connection timeouts are the same for all connections in the same state. For example, all TCP connections in the `ESTABLISHED` state (traffic has been seen in both directions) expire after the same period of inactivity (5 days by default), while TCP connections in the `SYN_SENT` state (only the SYN has been seen) expire much more quickly (2 minutes by default).

However, the timeout length be progamatically modified as desired for individual flows using the `netfilter-cttimeout` library on the Linux operating system. This allows an organization to programatically change how long various flows have until they are removed due to inactivity. For example, an organization may want to aggressively expire `SSH` connections, but not web traffic.

When a hard-timeout occurs, we clear the connection mark without affecting the connection state in the kernel, causing the next packet in the flow to trigger a query to the controller. Unless otherwise specified by the controller, no hard-timeout is specified. Instead, the use of soft-timeouts is favored. OpenFlow traditionally uses hard-timeouts to roughly approximate when a connection ends, since it lacks visibility into the state of the communicating end-hosts. However, our approach explicitly sends a notifications to the controller when a connection expires due to a soft-timeout. The SDN controller is kept current in our approach, without the extra overhead of forcing re-elevations to the controller.

5.5 Controller

The controller is a logically centralized application that runs within the organization's network. While we have created only a single SDN controller implementation, future users could create similar controllers in other languages and implementations, as happened with OpenFlow. The controller facilitates a publish-subscribe system, allowing applications to subscribe to events from host agents in certain subnets or even network-wide. When the controller receives an event from an agent, it passes that event on to all subscribed applications. The controller then aggregates the responses from all the applications and transmits the result to the host agent.

5.6 Compatibility with Non-Linux Hosts

While our initial implementation is tied to the Linux kernel, our approach 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 [43]. 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 [38] may provide the needed support for host agents, but further exploration is needed for conclusive results.

Other devices or operating systems may be unable to support a native host agent. However, we can provide many of the networking features used in the agents using a bump-in-the-wire solution. We installed our approach on a Raspberry Pi 1 Model B+ with instruction to provide forwarding between a directly connected host and the rest of the network. To test this configuration we used a host running Mac OS X Yosemite and a host running Windows XP. For both systems, we downloaded a file hosted by a virtual machine using the `wget` command, provided by `cygwin` for the Windows XP host. In both cases, the network connection was reported and handled correctly, confirming the ability to control the traffic flows just as could be done with a native solution. This approach allows for a plug-and-play style deployment for new devices. While this approach can operate the fine-grain access control features, it is limited to just network data; a smaller host-based agent could be used on partially supported operating systems to gather limited host context and inform the Pi during connections.

We note that this approach is a possible avenue for exploring the approach on systems without native support. However, given the logistical and capital costs, it is more suited for rapid prototyping. For organizations that need to support legacy hosts, such as embedded or "bring your own device" hosts, other strategies are likely to be more compelling. We describe these options in more detail in Section 6.

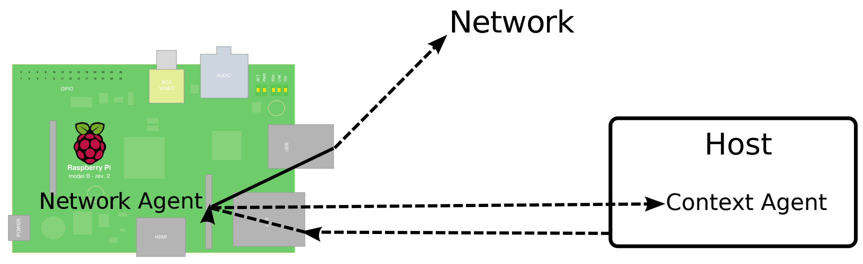

Figure 12: Non-Linux hosts can be supported via a bump-in-the-wire solution

5.7 Host Context

As our agent is running directly on the host, it is able to provide the controller with additional context for elevated packets. While we primarily focused on associating packets with the responsible application and user, we are also able to provide the on-screen text for the responsible application.

5.7.1 Gathering Basic Host Context

As an initial instantiation of the approach, we create an agent that extracts the associated process, including the path to the process's executable, and the information about the user and group of the process owner.

The host agent can be customized to gather context based on the type of operating system in use on the host. The agent may perform arbitrary pre-programmed commands to obtain information from system data structures (such as examining information in `/proc` on a Linux OS) or query any databases created by software on the host system. As a result, the agent can transmit any information gather by sensors on the host as long as those sensors provide an API for the agent to query. This allows organizations to customize what context will be provided to their controllers so that they can write policy on the controller to meet their needs.

5.7.2 Inferring User Intent

Interactions between users and systems are often performed using a graphical user interface (GUI). The user inputs some set of keystrokes and mouse actions to which the system responds. For example, users often spur applications into action by clicking buttons or submitting forms. The

GUI presents choices to the user in a well-structured way to make it easy for the user to understand the outcomes of any actions he or she takes.

By by collecting displayed text, mouse clicks, and other GUI interactions, we can gain insight into the high-level context of user applications. This insight may be useful when associated with network traffic. For example, it would be abnormal to see a connection to a network printer from an application that has not had a print button clicked nor any print dialog.

Given that a GUI is specifically designed to be easily understood by a user, we aim to systematically extract the information from the GUI and make it available for use by our SDN implementation. While the GUI may include a number of graphics, such as symbolic icons, the interface is often highly text focused and uses particular phrases intended to indicate possible actions a user may take. We take advantage of this design model to extract the text and leverage the graphical widget hierarchy to understand the relationships between text, buttons, and other on-screen containers (such as windows and dialogs).

In Figure 13, we provide an example of how graphical interfaces can be reused for establishing context for low-level actions. The left-side of the figure depicts the dialog box created by the Google Chrome web browser when the user selects to print the page. If the user clicks a button in the dialog, the process depicted on the right side of the figure will be initiated. The process will begin with an event-handler for the button being triggered. The system will then navigate the widget hierarchy to extract the text from the button, along with all text contained in the button's parent container (in this case, the dialog box) and the text of all other elements contained in that parent object, including label text and the text from other buttons in the dialog box. We then fuse this information with the low-level behavior of the process that created the activated button. We monitor all system calls that occur immediately after the button is clicked to identify possible causal links between the actions. We can then provide this information to access control systems. In this example, we may be able to tell the access control system that network traffic generated by process 3824 may be the direct result of the user clicking a button, with an indicated degree of confidence in the likelihood of that causal link.

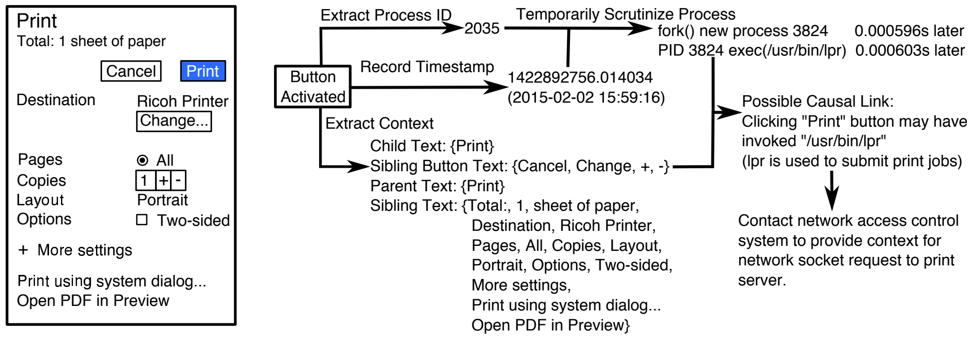
Print
Total: 1 sheet of paper
Cancel
Print
Destination
Ricoh Printer
Change...
Pages
All
Copies
1
+
-
Layout
Portrait
Options
Two-sided
+ More settings
Print using system dialog...
Open PDF in Preview
Button Activated
Extract Process ID
2035
Temporarily Scrutinize Process
fork() new process 3824 0.000596s later
PID 3824 exec(/usr/bin/lpr) 0.000603s later
Record Timestamp
1422892756.014034
(2015-02-02 15:59:16)
Extract Context
Child Text: {Print}
Sibling Button Text: {Cancel, Change, +, -}
Parent Text: {Print}
Sibling Text: {Total:, 1, sheet of paper, Destination, Ricoh Printer, Pages, All, Copies, Layout, Portrait, Options, Two-sided, More settings, Print using system dialog... Open PDF in Preview}
Possible Causal Link:
Clicking "Print" button may have invoked "/usr/bin/lpr"
(lpr is used to submit print jobs)
Contact network access control system to provide context for network socket request to print server.

Figure 13: We automatically extract contextual information surrounding user actions, such as button clicks, and fuse it with low-level analysis to inform access control systems.

We focus on instrumenting sensors and data collection in graphics libraries in our approach. This is a strategic decision: graphics libraries have well-defined abstractions and there are relatively few graphical libraries in use. These libraries have a set of API calls for inserting text into a widget along with a clear container hierarchy of child/parent relationships amongst widgets. This information is extremely useful for understanding a graphical interface. Further, since these libraries are heavily reused across applications, the instrumentation across a small number of libraries will yield coverage among a large number of user applications. Application designers have incentives to use these libraries, both to achieve higher productivity and to reduce the likelihood of errors when creating their own graphics manually.

We focus on the GIMP Toolkit (GTK+) graphics library in this project. GTK+ is one of the most popular graphics libraries for applications in the Linux operating system. Our future work will include instrumentation of other libraries, such as QT or WebKit, and integration with other traditional operating systems, such as Microsoft Windows and Apple OS X. However, we note that our approach can be generally applied to other libraries even if the implementation details vary.

By carefully choosing API calls in these libraries, we can extract the needed contextual information with trivial computational overheads. The libraries typically have functions to set text for an object container, such as a widget label. These functions are typically called once when the container is created and each time the text needs to be changed. This is in comparison to the functions used to render the objects, which may be frequently called as the container is manipulated, including window resizing or scrolling behavior. When we hook the text setting functions, we perform a simple memory copy to extract the text buffer supplied by the application. Likewise, we can hook the widget creation and linking events, such as setting a child or parent container, to record the memory addresses of these containers for tracking and reconstructing the hierarchy on demand. By using relatively lightweight operations, and by only invoking them upon GUI object creation and alteration, we can avoid noticeable performance overheads for the user while gaining all the information we need.

Once we have the information from the GUI, we can link it together by reconstructing the container hierarchy. Using the recorded memory addresses of the widgets extracted in the widget linking and text setting events, we construct a graph of elements. This allows us to use simple breadth-first searching to reconstruct the context surrounding buttons as they are activated. We further record the process ID of the application issuing the GTK+ calls, allowing us to link the window with the low-level system behavior associated with the process.

To connect the GUI-level events with the actions taken by the system, we log all system call behavior of each process using the Linux Auditing System. This allows us to efficiently record system call events as they happen. When a GUI event occurs, such as a button invocation in a process, we search for all events associated with that process ID immediately afterwards. We then present this information in an XML-encoded format to allow it to be consumed by security applications. In future work, we can provide a likelihood probability for the association. Such a probability can be constructed using historical associations (e.g., "in all past traces, when a button labeled "Print" was pressed, `/usr/bin/lpr` was invoked within $50\mu s$") or using text similarities (e.g., "Print" was on the button and the manual entry for `lpr` references the word 'print' `X` times"). However, in this work, we simply present all potential associations without indicating the probability of the link.

While our work focuses on automatically extracting the context for events, we discuss approaches to determine the likelihood of causal links and how to actionably use this information in Section 6.5. Such approaches could easily provide a plugin for our host agent.

5.8 Anti-Reconnaissance

By hindering an adversary's ability to perform network reconnaissance, we can limit its ability to spread within the network. We accomplish this by masking the network addressing information (MAC and IP addresses) at all point in which an adversary might observe them. The result is a moving-target defense system that prevents an adversary from spreading within a network without knowledge of internal DNS host names.

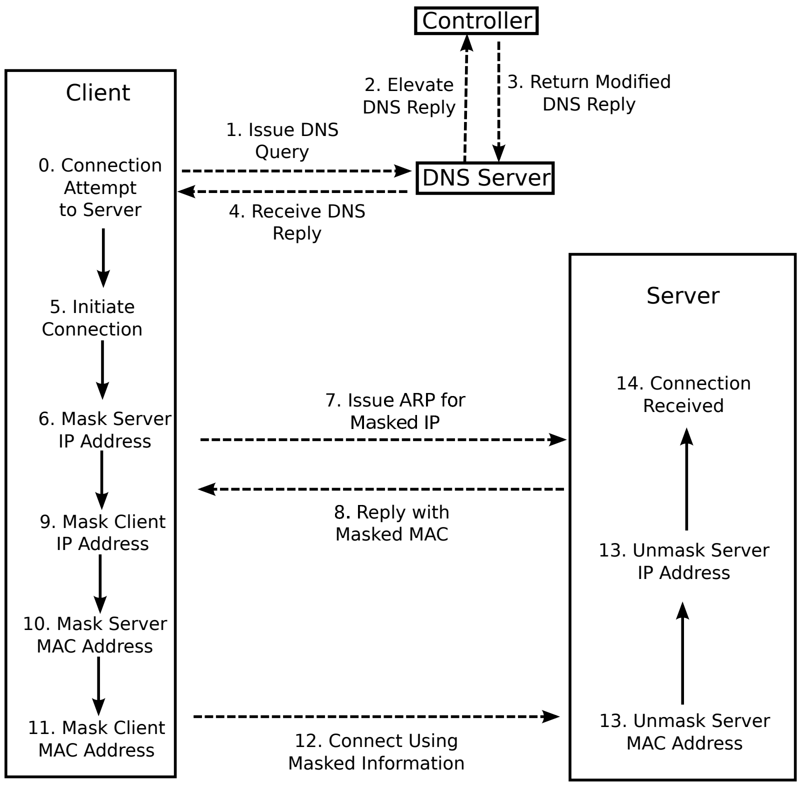
Controller
2. Elevate DNS Reply
3. Return Modified DNS Reply
Client
1. Issue DNS Query
DNS Server
0. Connection Attempt to Server
4. Receive DNS Reply
5. Initiate Connection
Server
14. Connection Received
7. Issue ARP for Masked IP
6. Mask Server IP Address
8. Reply with Masked MAC
9. Mask Client IP Address
13. Unmask Server IP Address
10. Mask Server MAC Address
11. Mask Client MAC Address
12. Connect Using Masked Information
13. Unmask Server MAC Address

Figure 14: Overview of the anti-reconnaissance process. For simplicity, the elevation of the connection to the controller has been omitted.

Our approach operates by utilizing synthetic addressing information in place of the real addressing information. For simplicity, the synthetic information can be considered to be a randomly chosen valid value. The `iptables` and `ebtables` tools are used to mask/unmask the network and link-layer addressess, respectively. Note that `ebtables` only works on bridged interfaces. Normal hosts to have a bridged interface configured to contain just their single normal interface, allowing `ebtables` to be applied.

Our system enables us to intercept DNS responses from the organization's DNS server. The anti-reconnaissance host application running on the controller then replaces the real IP address with a synthetic one. This will be the IP address that an adversary monitoring a compromised host's connections will see. The modified DNS response is then returned to the DNS server and is then forwarded to the host.

Both the querying client and the server indicated in the DNS response receive several commands before the DNS response is returned to the DNS server. First, both receive a command similar to the one below, which causes them to respond to ARP requests for the selected synthetic IP address with the selected synthetic MAC address.

```
ebtables -t nat -A PREROUTING -p arp --arp-ip-dst 10.0.0.100 --arp-opcode Request
-j arpreply --arpreply-mac 52:54:00:a2:69:10
```

After that, the client receives commands to mask the source and destination IP and MAC addresses of all packets in the connection. The first two commands below will mask the source and destination IP addresses, while the second two will mask/unmask the MAC address.

```
iptables -A OUTPUT -t nat -p tcp -o br0 -d 10.36.1.11 --dport 12345 -j DNAT
--to-destination 10.36.1.110:54321

iptables -A POSTROUTING -t nat -o br0 -d 10.36.1.110 -j SNAT --to-source
10.36.1.100

ebtables -t nat -A POSTROUTING -p ipv4 --ip-source 10.36.1.100 --ip-destination
10.36.1.110 -s 52:54:00:a2:69:90 -j snat --to-source 52:54:00:a2:69:10

ebtables -t nat -A PREROUTING -p ipv4 --ip-source 10.36.1.110 --ip-destination
10.36.1.100 -d 52:54:00:a2:69:10 -j redirect
```

The server will receive similar commands, enabling it to unmask the destination IP address and to mask/unmask its MAC address.

There are two possible viewpoints the adversary can have in the network. In the first, the adversary has managed to gain user-level access to a legitimate host, and is passively monitoring the connections to and from the host. This can be done without root permissions using a tool such at `netstat`. In the second, the adversary can view traffic between other hosts, allowing it to see all the addressing information in those packets. This may be the case if machines not administrated by the organization can connect to or observe the network, even if they are otherwise restricted in whom they can connect to.

In the first case, the adversary only gains limited useful information. The host never knows the real addressing information of its counterpart. There is no way for the adversary to learn the real addresses by monitoring the host. However, this is contingent upon the use of DNS. Once the host application has the IP address it is going to connect to, the connection state is established in the kernel before we have a chance to manipulate it. Altering this would require per-application support or kernel modifications, which are beyond the scope of our SDN agent implementation. All the addressing information can still be masked in the network however.

In the second case, the adversary does not gain any useful information about the hosts in a connection by observing that connection. Both the source and destination addressing information is masked and is not usable outside of that connection. However, by observing the DNS traffic, it can gather host names. These would allow the adversary to successfully connect to other hosts, albeit not necessarily specific ones if DNS load balancing is used. However, it is trivial to write a controller application that masks the host names before they leave the host and unmasks them once they arrive at the DNS server. As the communication between host agents and the controller is encrypted, the adversary will not be able to view the real DNS host name in the network.

By masking the network addressing information in this fashion, we can prevent an adversary from acquiring information necessary to create connections on their own simply by observing either the network traffic or the connections of a legitimate host. The system acts as a network moving target defense, keying off of DNS host names and preventing new connections that do not engage the system via a DNS lookup.

6 Discussion

We now consider how the approach can be leveraged to provide more security and/or control and how the approach would be deployed within an organization and the functionality of hosts when remote to the organizational network.

6.1 Fine-grained Control and Visibility

When fine-grained flows are used in an SDN implementation, every new flow is elevated to the controller, regardless if the destination in in the same subnet or VLAN. This gives the controller precise situational understanding in which it is aware of all flows creates in the network. The controller can leverage this to search for network wide trends, construct real-time communication graphs, and log information for later forensic analysis. Both the visual understanding of the network [33] and the communication graph [22] can be useful to deal with network attacks.

Fine-grained rules also allow traffic to be transparently forwarded through network security middleboxes, such as intrusion detection systems or firewalls, on a per-flow basis. Such middleboxes are analogous to the concept of a waypoint, described by Casado *et al.* [14]. This allows us to subject each flow to a varying amount of scrutiny, instead of having make decisions about groups of flows as is traditionally the case.

6.2 Contextual Policies

Various contextual languages for high-level policy specification, such as `POL-ETH` [14], Flow-based Security Language (FSL) [26], and Flow-based Management Language [27], are available to network operators. While these policy languages are conducive to easy formal analysis, they currently cannot account for multi-user systems. While prior work has proposed such distinction in the future [41], our attempt is the first to actually do so, to the best of our knowledge. Furthermore, our implementation is able to provide additional host context not considered by some of these prior works.

We illustrate our approach's potential by giving three example policies our approach can enforce that prior work could not.

6.2.1 Example Policy: Executable and User Context

In our initial implementation, we developed a plugin that gathers the following host context for each new connection initiated or received by the host:

1. The basic flow tuple
2. The absolute path of the executable and arguments
3. The username of the process owner
4. The primary group of the process owner
5. If all ancestor processes are located in administratively-controlled directories

Even just this small set of contextual information can be used to enforce a powerful policy that was previously impossible using switch-based SDN infrastructure.

```
# Groups
desktops = [“griffin”,“roo”];
server = [“http_server”];
students = [“bob”,“bill”];
profs = [“plum”];
%%
# Processes
browserApp = [“/usr/bin/chrome”,“/usr/bin/firefox”];
browserExt = [“–no-javascript”];
webServerApp = [“apache2”];
%%
# Rules
# Allow private to connect to server computers
[(hsrc=in(“private”)∧(hdst=in(“server”))] : allow;
# No restrictions on desktops communicating with each other
[(hsrc=in(“desktops”)∧(hdst=in(“desktops”))] : allow;
# Only allow professors running Chrome or Firefox
# on desktops to connect servers running Apache
# if JavaScript is disabled
[((hsrc=in(“desktops”)∧hdst=in(“server”))
∧(user=in(“profs”))
∧(appsrc=in(“browserApp”)∨appdst=in(“webServerApp”))
∧(appext=in(“browserExt”)] : allow;
[]: drop; # Default-off: by default drop flows
```

Figure 15: Ethane’s [14] extended sample policy using context information.

In Figure 15, we show a simplified example policy originally given in Ethane [14]. We note the additional contextual information available under our approach that was beyond the capabilities of Ethane in bold. While the original policy was only able to restrict communication at a host-level granularity, we can be as specific as only allowing web traffic from an approved browser to an Apache web server instance. Furthermore, we can restrict traffic based on browser options, such as requiring that JavaScript be disabled as an additional security measure.

However, even though our approach can gather nuanced information such as the user or group ID owning an application, being implemented in end-hosts does restrict some policy options. For example, we cannot enforce rules based on a particular physical switch port. Similar rules can be generated by rewriting the originals to apply to the source and destination end-hosts instead of intermediate points in the network infrastructure.

6.2.2 Example Policy: Graceful Degradation for Mission Continuity

Organizations must make strategic choices about dealing with compromised hosts on their networks. While it may be ideal to take compromised systems off the network and restore from backups from a security perspective, this can interfere with the desire to preserve forensic information for the purpose of prosecution or counter-intelligence [5].

In traditional networks, the organization can generally only decide between completely isolating a compromised host or allowing it to communicate arbitrarily. If the compromised machine is involved in mission-critical services, it may not be feasible for the organization to halt the system for any period of time but failing to contain the compromise can be an unacceptable risk. Our approach allows for fine-grained policies which allow the organization to flexibly respond when a host is compromised.

Consider an example in which a some host, A, is compromised. A legitimate client application on A needs to regularly contact a server within the organization's network. However, as A has been compromised, the organization wants to restrict the traffic coming from A to contain the compromise. Our approach allows the organization to write a policy that only allows the legitimate application to initiate network communication, denying all other attempts. Furthermore, the policy can be enforced on the part of the organization's servers as well. This allows the organization to contain the compromise without disrupting the application running on the compromised machine. By utilizing host context, we can enforce such a strict policy with limited collateral damage where other approaches, such as OpenFlow or network firewalls, fundamentally cannot.

6.2.3 Example Policy: Server Load Balancing

In addition to security policies, our approach allows us to specify policies aimed at improving network performance. As an example, we consider a policy that utilizes host context to make load balancing decisions.

An established, lightweight, load balancing technique is round robin DNS [13]. However, while simple, round robin approaches don't account for actual load on end-hosts. In scenarios where network connections do not all require the same amount of resources, some servers can end up with a disproportionate workload. As such, dynamic load balancing techniques that do account for server load are superior to simple static approaches [48]. Instead of relying on proprietary systems to gather this information, we can achieve dynamic load balancing via a policy on our logically centralized controller.

To do so, we can create a plugin to be installed on servers' SDN agents that gathers the desired load information, such the CPU and memory usage, application thread count, or I/O activity. The load balancing application running on the controller can consider such context, as well as network history, to select which server handles each request in a dynamic fashion. Furthermore, the client's host agent may be able to provide context that can determine of the request is likely to be CPU or I/O bound. This allows an organization to evenly balance traffic between servers, using whichever metric(s) they choose.

6.3 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. Our approach interacts 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 deploy, we degrade to the limitations of a traditional network infrastructure and lack insight into the traffic between the hosts. However, when one of the two communicating hosts deploys, the situation becomes. Finally, having a single host participating in a flow is analogous to an external host communicating to an internal host as described in Section 4. 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, essentially acting as an easily multiplexed equivalent of our bump-in-the-wire approach. 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.

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.

6.4 Hosts Roaming on Remote Networks

In many cases, an organization may have mobile hosts, such as laptop computers, that may be used on networks other than the deploying organization's network. Since the agent is simply a privileged application on the host, the organization may customize the agent's response to a remote network. If the organization desires it, the agent may recognize that it is remote to the organization when it attempts to reach the controller and automatically establish a VPN connection to the organization and route all the traffic into the organization by default, allowing the organization's controller to continue managing the device. Other organizations may simply allow the host to communicate as if the agent were not present when visiting remote network. Importantly, organizations have robust control over the network behavior of the host even when the host is remote. This is in contrast to the OpenFlow approach, which only affects hosts on the organization's network while they are on the network.

Furthermore, our system can be utilized to provide SDN functionality to hosts within a home network, with a controller placed in the cloud. This would allow for a residential network to take advantage of services such as intrusion detection, proxying, and DNS blacklisting, without having to configure or maintain such services themselves. Such services could be provided as applications running on the controller in the cloud, where the controller itself could be provided as a service by a third party.

6.5 Determining the Likelihood of Causal Links for User Intent

We discussed how we can provide the text within an application's GUI as part of the context for packets. However, while such context is easily gathered, other approaches could provide much more useful context. A variety of techniques, from simple heuristics to sophisticated modeling, can be used to determine the likelihood of links between user actions and the low-level system manifestations of each.

The most basic approach is to use event timing. If an event in a given process, B, occurs immediately after another event, A, in that same process, then B may have been caused by A. This naïve approach may yield false associations due to coincidence, such as independent events that happen to occur in close temporal proximity. However, when combined with historical runs of the process, a more robust finding may occur. If event B occurs if and only if A occurs immediately beforehand, then the link between B and A may be stronger. Such relationships can be formally analyzed using Markov modeling to calculate probabilities in a more robust manner.

A stronger approach may use the text extracted more directly. The text from context surrounding a button event may be translated into a simple bag of words. Likewise, any documentation surrounding the underlying system events, such as manual pages for executed applications or system calls, can be placed in a separate bag of words. After extracting stop words, terms from the two bags can be compared. If numerous relatively rare terms, from the collection of all command documentation, appear in both bags, the likelihood probability may be increased. More sophisticated topic modeling techniques, such as LDA [12], can be used to determine the topic of the context and that of the documentation of system invocations. These tools can be used to calculate the topic intersections and formally analyze the likelihood they are related.

The most naïve approaches may be used immediately with reasonable accuracy while the more sophisticated approaches can be developed and refined to offer high accuracy to users [34]. If user intent can be accurately inferred and linked to network traffic, then we may be able to write policies that are capable of disallowing traffic even from authorized applications if those applications are behaving abnormally. For example, such a policy could be used to hinder malware that takes over a legitimate application.

6.6 Anti-Reconnaissance

Our anti-reconnaissance system can be implemented in various ways to achieve different effects. An organization can make decisions to achieve their own particular goals.

6.6.1 Handling External Destinations

As the DNS reply is forwarded to the controller from the host, this approach works even for traffic with a destination external to the organization. In that event, the controller simply needs to instruct the agent on the last host that the traffic will pass through to transform the synthetic information to the global information. If desired, the source host can be instructed to use the global information for network communication, while still using the randomized IP address in the modified DNS response. This hides the real destination IP address from an adversary monitoring a compromised host, but not one that can view the organization's network traffic.

6.6.2 Selecting Synthetic Values

While randomizing addressing information hinders an adversary's reconnaissance attempts, we must take care to avoid hindering the organization's understanding of their own network. Before an administrator can make sense of the network traffic, the synthetic information must be converted back to the real information. To avoid having to search a log to find the appropriate mapping, the synthetic address information can be generated by encrypting the address information, in addition to a nonce. The encrypted value would appear random and can easily be reversed via decryption without having to consult state at the controller.

Of course, we must take care that the encrypted addresses are still valid. For example, encrypting the IP address `10.1.1.100`, contained in a `10.1.0.0/24` subnetwork is unlikely to result in an IP address that is valid in that subnet. Instead, just the last two bytes could be encrypted. Furthermore, things such as the encryption algorithm's block size must be considered to reduce the possibility of collisions.

Beyond how the synthetic values are generated, an organization can also tune when they are generated. Specifically, new values can be generated for every new connection made or they can be generated in fixed time intervals. The former provides more security, as the same information will be different even if used by two hosts at the same time. The latter requires less overhead, reducing the state needed on the host as well as the amount of communication with the controller.

Ultimately, the organization is free to choose the balance that suits it best.

6.6.3 IPv6 tunneling

When creating synthetic addresses, the system must avoid addresses that are already in use. Combined with the comparatively small IPv4 space, particularly when restricted by subnetworks, a system that generates a new synthetic IP addresses for each new connection can run the risk of exhausting available IPs. To address this, the system can transparently encapsulate traffic via an

IPv6 tunnel. The vast number of IP addresses available in the link-local (`fe80::/64`) address space guarantees that even the most prolific networks are extremely unlikely to run out of IP addresses available for use by the anti-reconnaissance system. A similar technique is used by Dunlop *et al.* [18] for a distributed moving target defense system.

6.6.4 Targeted Deception

Our basic goal is to prevent the adversary from learning useful information about the organization's network. As currently presented, the system hinders the adversary's ability to locate hosts and to even understand how many hosts are in the network. However, the system can also be used to deliberately mislead the adversary as well. Honeypots are a common example of targeted deception, where an adversary believes they have compromised a real host while the organization monitors the actions that the adversary takes. By leveraging our agents, we can gain many of the benefits of a honeypot on a real host once it is discovered that that host has been compromised.

Consider the IP address in the controller-modified DNS reply. Its sole purpose is to provide an address for the establishment of connection state on the host that cannot be associated with the real destination. The address is never used for actual communication, it simply is matched against for the transformation to the synthetic information. We can leverage this to make connections appear to be to a completely different host.

Rather than providing a random address in the DNS reply, the controller can select an actual address, albeit one that does not correspond to real destination. Since the address with be transformed to its synthetic counterpart before any network communication takes place, the host corresponding to the actual address is never contacted. If the adversary can be expected to know some valid IP addresses, this can lead them to false conclusions. Furthermore, this can make external traffic appear to be internal, and vice versa.

Once a host has been discovered to be compromised, we can exercise this approach to give the adversary the impression that it is connecting to its intended destinations. The actual connections can instead be directed to an arbitrary destination without the adversary's knowledge, provided that the destination provides the intended service. For example, an adversary's `SSH` attempts could be redirected to a honeypot providing the `SSH` service. Furthermore, note that this does not require that the anti-reconnaissance system be employed, this approach can be engaged completely independently.

7 Evaluation

To demonstrate and evaluate our approach, we implemented it 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 MB of RAM. The network controller is allocated two cores and 2048 MB 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-grain 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 general good security practice [10,35].

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.

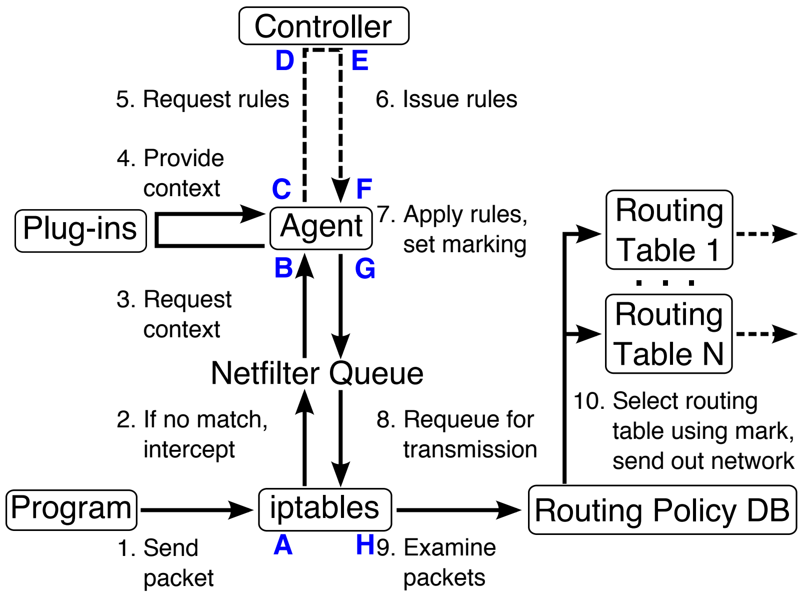

Figure 16: Overview of kernel and agent communication. Dashed lines represent network traffic while solid lines represent intra-system communication. The bold, blue letters indicate measurement points for the performance evaluation in Table 1.

7.1 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 [32, 52] 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 located in administratively controlled paths, if the process is being run from a window manager, if the process is being executed by a privileged user, and if the process has been executed from a shell.

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 16. We performed these timings on one of the hosts and the controller using `ovs-benchmark`'s [2] batch mode to create $1,000$ sequential connections. For each connection, the policy presenting in Section 6.2 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 over-all end-to-end timings with all intermediate timing samples disabled. We present the results of the timing experiment in Table 1.

We show these results in Table 1. We see that our SDN approach incurred a median of just under 17 milliseconds, with a significant portion of that being devoted to gathering the host context. Further, this overhead is only incurred at the beginning of the network connection and thus may have little impact on actual applications. In the case of TCP, the overhead takes place during the handshake, before slowstart comes into effect.

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. The gathering of host context makes up a majority of the overall overhead, with the actual elevation to the controller also taking a fair portion of the time The marking, which is currently done by executing an `iptables` in a subprocess can be implemented using the `netfilter-conntrack` in the future, which should reduce the time at that step as well. The gathering of various host context is done sequentially, and could be parallelized in the future.

	Fig. 16 Steps		Median	Std. Dev.
Component Description	Start	End	(ms)	(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	A	H	16.72	1.403

Table 1: Component-wise characterization of latency overheads over 1,000 TCP connections. Columns 2 and 3 correspond to the bold, blue letters in Figure 16.

Num. Hosts	Packets/s	Median RTT (ms)	Std. Dev. (ms)
2	27.4	33	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
2 OF Hosts	243.3	4	1.23

Table 2: Round trip times with each host transmitting 1,000 packets.

7.2 Scalability of the Controller and Agents

We also explore the scalability of our approach, from the perspective of a host. In these experiments, we vary the number of simultaneously communicating hosts from two to fourteen, in even increments, with an additional experiment using 28 hosts. In all experiments, each host is pinned to a single core, with the controller being pinned to two cores. In all experiments except for the last, each host was the sole host pinned to that particular core. Each host used `ovs-benchmark`, run in batch mode, to rapidly create $1,000$ sequential TCP requests to another host. All hosts are paired with another, with each host both receiving and creating the same number of requests such that they are all under an equal load. As none of the hosts are listening for TCP connections, they all respond to requests with a `TCP+RST`, allowing the sender to calculate the RTT, instead of the time for a 3-way handshake. Both the `TCP+SYN` from the sender and the `TCP+RST` from the receiver are elevated to the controller. We also record the number of new flows created per second by a single host. We run the additional experiment, using 28 hosts, to verify that the number of cores available on our testing infrastructure is the primary limiting factor. We show the results of these experiments in Table 2.

Under our approach, the maximum request rate for new connections is approximately 25 per second, with a median RTT time ranging between 34 ms and 41 ms. The lower RTT values are slightly more than twice the overall time taken by the agent on a single host ($\approx$ 16.8 ms) and does not take into account transmission delay. While the RTT does slightly increase with the number of hosts involved in the experiment, the increase does not suggest that the number of hosts is the deciding factor. In the 28 host experiment, in which each core has two hosts pinned to it, the RTT is approximately double that of the 14 host experiment, with the requests per second being approximately half as much. This confirms our expectation that the number of cores available on our testing infrastructure is the influencing factor for larger number of hosts.

Overall, we saw roughly 350 new requests per second (with 14 hosts) where each host was responsible for approximately 25 requests per second. This is a significant overestimate of the number of new flows per second from each host as compared to Ethane [14]. The heaviest load seen among the three different network data sources used in the paper was, on average, less than 3 new flows per second for each host in the worst case. Unlike switch-based SDN implementations, such as OpenFlow, all of our per-flow state is kept in the hosts, removing the scalability constraints from the network infrastructure. The only constraints on the number of flows a host can create per second, as well as the overall number of simultaneous flows, are the computational resources available to that host and the time it takes for each flow to go through the elevation process. As our controller only takes around 5 μs processing each packet, it can theoretically handle around 200,000 elevation requests per second in the network. While we expect lower values in actual networks, we do note that the POX Python controller can only handle approximately 35,000 requests per second [19].

7.3 Evaluating Policy Enhancements on Security

In Section 6.2.1, we detail a policy in which the controller only allows applications owned by regular users to create network connections if the process is installed to a administratively controlled directory (e.g., `/usr/bin/`). We now evaluate whether such a policy would be able to hinder persistent user-level malware.

Our first experiment uses the toy Linux remote access tool (RAT) n00bRAT [3], which allows an adversary to connect to the compromised machine and perform actions from a shell, such as exfiltrating the contents of `/etc/passwd`. We modified the malware's source code to listen on a non-privileged port, instead of port 80, to meet our restriction to user-level compromises. Note that our threat model also prevents the adversary from executing any shell commands that require root. In our experiment, we model two infection vectors. In the first, we have a user download the malware as an email attachment, modeling a phishing attack. In the second, we infect the host by using Metasploit to exploit a vulnerable version of the Firefox web browser and subsequently cause the compromised browser to lauch the malware, modeling a drive-by download. In both cases, our policy prevented n00bRAT from communicating with the adversary.

In the experiment, where the malware was launched as an attachment from a popular email application, n00bRAT began to run as a separate process from the user's attachment folder. As the folder is owned by the regular user, n00bRAT registered as a user-installed program and the policy subsequently denied all connections both to and from the program. This illustrates that the policy can deny both connections originating from and destined to malware, regardless of the location of the other end of the connection.

In the experiment modeling a drive-by download, we used the exploit noted in CVE-2013-1710 via Metasploit to obtain a remote shell. As the exact exploits a vulnerability in Firefox, the remote shell starts as a thread within the web browser. As Firefox was an administratively installed program, the adversary was able to transfer the malware to the host. However, since Firefox was executed by the user, the adversary cannot transfer the malware to an administratively controlled directory. While the adversary is able to launch the malware in the background, the malware registers as user-installed and is denied network connectivity. Furthermore, the adversary will only retain network access to the machine as long as Firefox is running. Even attempting to run the malware via a Cron job will not enable the adversary to gain persistent connectivity, as the malware will not be able to access the network. Once Firefox is halted, the adversary will no longer be able to connect to the host.

Minimum	Median	Average	95th Percentile	Maximum	Standard Deviation
40	90	112	240	942	77

Table 3: Overhead time in μs over 104,004 data points after excluding 284 for being greater than 2 times the mean.

7.4 Capturing Text

To determine the performance overheads of our monitoring, we focused on the amount of time spent processing the logging routines we added, since these are the only overheads that could affect the user. Using the `clock_gettime` function, we recorded the system's timestamp immediately when our function began and again just before the function returned, recording the difference as the function's running time.

We ran some simple tests on an Ubuntu 12.10 virtual machine with 1GB of RAM and three 2.6 GHz cores available. In our simple tests, our logging function was invoked $104,004$ times. We excluded 284 outliers which were over two times the standard deviation from the mean; they were likely due to OS scheduling quirks rather than actual computational run-times. We summarize these results in Table 3. In general, each function call returned quickly, with a median of only 90 μs for each function. Given the imperceptible median and average overheads associated with this monitoring, we believe such monitoring can be incorporated into modern desktop systems without negatively affecting the system's usability.

8 Future work

There are several considerations to be made in future work.

8.1 Unreliable Context and Hosts

Currently, the threat model precludes a host agent being compromised and all the host context gathered is assumed to be reliable. This is largely due to the fact that, with the exception of the text extracted from GTK, all the host context requires root to modify. However, in the real world root compromises do happen. While techniques such as virtualization can be employed to protect the host agent, such techniques do incur overhead that an organization may find undesirable. Furthermore, the host context gathered by the agent will be much more difficult, if not impossible, to protect in such a fashion.

Instead, future work will consider how to handle possibly unreliable host agents and/or host context. Ideally, even in the event of root compromises among hosts, the network will still be able to provide some subset of security guarantees and/or be able to detect such a compromise. For example, if a server reports an incoming connection from a client that has not notified the controller due to a compromise suppressing the message, the controller may take the client off the network and notify and administrator.

8.2 Proxying and Mobile Devices

Our approach requires either that hosts run a software agent or utilize a bump-in-the-wire solution. However, neither option currently works for mobile devices. One route to consider is deploying the agent in a bump-in-the-wire behind wireless access points. However, this is infeasible when the mobile device is not within the organization or when the device is using a mobile network. Other work has looked at incorporating middleboxes in mobile traffic [44]. Similar approaches will be considered for our SDN implementation in the future.

8.3 Interaction with OpenFlow

While originally designed to address issues that OpenFlow is not well-suited for, our host-based implementation does not incorporate all the low-level functionality of a switch-based implementation. For example, we are not able to programatically address link failures in the network or provide access control for switch ports. We will explore the efficacy of combining OpenFlow and our implementation within a single network, with OpenFlow providing the coarse network transit functionality and our implementation handling the fine-grained access control and context components.

9 Concluding Remarks

Our host-based SDN implementation allows for scalable fine-grained flow-based monitoring for enterprise networks. We created a protoype implementation and evaluated it both on real physical hosts and in a virtual network. We found that our approach incurred acceptable overheads that are unlikely to be noticed by end-users. Using our system, organizations can gain the benefits of SDNs without having to significantly alter their existing network infrastructure, and have the option of incrementally deploying the system. The logically centralized controller gives operators a better picture of their network, and the host context provided by our system allows nuanced policies to be enforced. Information, such as individual users and applications responsible for network traffic, can be identified and utilized. Overall, this enables for richer and more powerful organizational policies.

References

[1] Openflow specification version 1.3.0, June 2014.

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

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

[4] Moheeb Abu Rajab, Fabian Monrose, and Andreas Terzis. On the impact of dynamic addressing on malware propagation. In *ACM Workshop on Recurring Malcode*, pages 51–56. ACM, 2006.

[5] Frank Adelstein. Live forensics: diagnosing your system without killing it first. *Communications of the ACM*, 49(2):63–66, 2006.

[6] Ehab Al-Shaer. Toward network configuration randomization for moving target defense. In *Moving Target Defense*, pages 153–159. Springer, 2011.

[7] Spiros Antonatos and Kostas G Anagnostakis. Tao: Protecting against hitlist worms using transparent address obfuscation. In *Communications and Multimedia Security*, pages 12–21. Springer, 2006.

[8] JV Antrosiom and Errin W Fulp. Malware defense using network security authentication. In *Information Assurance, 2005. Proceedings. Third IEEE International Workshop on*, pages 43–54. IEEE, 2005.

[9] Sean Barnum. Standardizing cyber threat intelligence information with the structured threat information expression (stix). *MITRE Corporation*, page 11, 2012.

[10] Steven M Bellovin. Security problems in the TCP/IP protocol suite. *ACM SIGCOMM Computer Communication Review*, 19(2):32–48, 1989.

[11] Wilson Naik Bhukya, Suneel Kumar Kommuru, and Atul Negi. Masquerade detection based upon gui user profiling in linux systems. In *Advances in Computer Science–ASIAN 2007. Computer and Network Security*, pages 228–239. Springer, 2007.

[12] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[13] T. Brisco. DNS Support for Load Balancing. RFC 1794 (Informational), April 1995.

[14] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, August 2007.

[15] Cisco, Inc. VLAN security white paper. `http://www.cisco.com/en/US/products/hw/switches/ps708/products_white_paper09186a008013159f.shtml#wp38986`, 2014.

[16] Weidong Cui, Randy H Katz, and Wai-tian Tan. Design and implementation of an extrusion-based break-in detector for personal computers. In *Computer Security Applications Conference, 21st Annual*, pages 10–pp. IEEE, 2005.

[17] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[18] Matthew Dunlop, Stephen Groat, William Urbanski, Randy Marchany, and Joseph Tront. MT6D: A moving target IPv6 defense. In *Military Communications Conference, 2011-MILCOM 2011*, pages 1321–1326. IEEE, 2011.

[19] David Erickson. The beacon openflow controller. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, pages 13–18. ACM, 2013.

[20] Glenn A. Fink, Vyas Duggirala, Ricardo Correa, and Chris North. Bridging the host-network divide: Survey, taxonomy, and solution. In *Conference on Large Installation System Administration*, LISA '06, pages 20–20, Berkeley, CA, USA, 2006. USENIX Association.

[21] Prashant Garimella, Yu-Wei Eric Sung, Nan Zhang, and Sanjay Rao. Characterizing vlan usage in an operational network. In *SIGCOMM Workshop on Internet Network Management*, pages 305–306. ACM, 2007.

[22] Carrie Gates, Michael Collins, Michael Duggan, Andrew Kompanek, and Mark Thomas. More netflow tools for performance and security. In *USENIX Conference on System Administration*, LISA '04, pages 121–132, Berkeley, CA, USA, 2004. USENIX Association.

[23] 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.

[24] Joel Goodman, Randall Seed, and Peter Kiefer. Dynamic subnets for sensor networks. In *IEEE Vehicular Technology Conference (VTC)*, September 2004.

[25] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *USENIX Security*, 2008.

[26] Timothy Hinrichs, Natasha Gude, Martın Casado, John Mitchell, and Scott Shenker. Expressing and enforcing flow-based network security policies. 2008.

[27] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *ACM Workshop on Research on Enterprise Networking*, WREN '09, New York, NY, USA, 2009. ACM.

[28] IBM. OpenFlow: The next generation in networking interoperability. `http://public.dhe.ibm.com/common/ssi/ecm/en/qcw03010usen/QCW03010USEN.PDF`, 2011.

[29] 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.

[30] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. OpenFlow random host mutation: Transparent moving target defense using software defined networking. In *Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 127–132, New York, NY, USA, 2012. ACM.

[31] M. St. Johns. Identification protocol. IETF RFC 1413, February 1993.

[32] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[33] Kiran Lakkaraju, William Yurcik, and Adam 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.

[34] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: using system-centric models for malware protection. In *ACM Conference on Computer and Communications Security*, pages 399–412. ACM, 2010.

[35] Christopher Low. ICMP attacks illustrated. *SANS Institute URL: http://rr. sans. org/threats/ICMP_attacks. php (12/11/2001)*, 2001.

[36] Long Lu, Vinod Yegneswaran, Phillip Porras, and Wenke Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *ACM Conference on Computer and Communications Security*, pages 440–450. ACM, 2010.

[37] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

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

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

[40] Richard Mortier, Tom Rodden, Tom Lodge, Derek McAuley, Charalampos Rotsos, Andrew W Moore, Alexandros Koliousis, and Joe Sventek. Control and understanding: Owning your home network. In *Communication Systems and Networks (COMSNETS)*, pages 1–10. IEEE, 2012.

[41] Jad Naous, Ryan Stutsman, David Mazieres, Nick McKeown, and Nickolai Zeldovich. Delegating network security with more information. In *ACM Workshop on Research on Enterprise Networking*, pages 19–26. ACM, 2009.

[42] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: dynamic access control for enterprise networks. In *ACM Workshop on Research on Enterprise Networking*, pages 11–18. ACM, 2009.

[43] OpenBSD Developers. Pf: The OpenBSD packet filter. `http://www.openbsd.org/faq/pf/`.

[44] Ashwin Rao, Justine Sherry, Arnaud Legout, Arvind Krishnamurthy, Walid Dabbous, and David Choffnes. Meddle: middleboxes for increased transparency and control of mobile traffic. In *ACM Conference on CoNEXT Student Workshop*, pages 65–66. ACM, 2012.

[45] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 1975.

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

[47] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*, pages 298–307. ACM, 2004.

[48] Sandeep Sharma, Sarabjit Singh, and Meenakshi Sharma. Performance analysis of load balancing algorithms. *World Academy of Science, Engineering and Technology*, 38:269–272, 2008.

[49] Jeffrey Shirley and David Evans. The user is not the enemy: Fighting malware by tracking user intentions. In *Workshop on New Security Paradigms*, pages 33–45. ACM, 2009.

[50] Craig A Shue, Andrew J Kalafut, Mark Allman, and Curtis R Taylor. On building inexpensive network capabilities. *ACM SIGCOMM Computer Communication Review*, 42(2):72–79, 2012.

[51] Peter Stephenson. The application of intrusion detection systems in a forensic environment. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2000.

[52] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[53] Twisted Framework Developers. Python twisted framework. `https://twistedmatrix.com/`.

[54] David Whyte, Evangelos Kranakis, and Paul C van Oorschot. DNS-based detection of scanning worms in an enterprise network. In *NDSS*, 2005.

[55] Hao Zhang, William Banick, Danfeng Yao, and Naren Ramakrishnan. User intention-based traffic dependence analysis for anomaly detection. In *Security and Privacy Workshops (SPW)*, pages 104–112. IEEE, 2012.