April 2016

Enhancing ACLs with Host-Context

Brett Lowell Ammeson
Worcester Polytechnic Institute

Julian Reed Moore
Worcester Polytechnic Institute

Michael Wallace French
Worcester Polytechnic Institute

Enhancing ACLs with Host-Context

Enhancing ACLs with Host-Context A Major Qualifying Project Report
Submitted to The Faculty
of the

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the

Degree of Bachelor of Science in Computer Science

by

Brett L. Ammeson

Michael W. French

Julian R. Moore

Approved by:

Craig A. Shue

Abstract

WinSight is a distributed firewall and network monitoring system capable of considering packets' host context when making flow decisions and is developed for Windows 7. With WinSight, machines on a given network have an agent installed which reports incoming and outgoing packet flows to a controller, which sends a response if the flow is deemed safe. Machines with WinSight will not accept a packet flow unless the controller approves that flow. To increase defense against internal network threats, such as worms and compromised machines, we developed both an agent and a controller which follows a popular standard called OpenFlow. Our testing showed WinSight is able to successfully block traffic based on context data and deep packet inspection with a moderate performance impact, with the first packet of each flow most affected. There were also rare, yet significant delays when reinjecting packets into the host's network stack.

Contents

1 Introduction

Networks, especially enterprise networks, are a vital component of communication. The importance of networks brings the motivation to attack them. It is possible to disrupt or halt network services through denial of service (DoS) and malware attacks. These attacks can come externally from the Internet or internally from a machine on the network. In either case, the severity can vary from minor inconveniences to catastrophic data leaks and system failures.

To combat these attacks, systems like personal firewalls, distributed firewalls, and network access control lists (ACL), have been created. Broadly speaking, these systems monitor the traffic on the network to prevent unauthorized and malicious packets from circulating. These systems can deter and halt malicious traffic on the network, but they are limited. The information they draw from to analyze traffic is often limited to networking data, such as source and destination addresses. While this is useful for blocking known exterior threats, it is less useful in blocking unknown interior threats. For example, if a trusted machine becomes compromised by a worm, most firewalls lack the sophistication to detect and block the malicious traffic without human intervention.

To enable and expand malicious traffic detection and prevention, we present WinSight. WinSight combines host context extraction with OpenFlow control for end hosts, creating a distributed firewall with simple whole-network monitoring. To realize WinSight on the Windows 7+ platform we develop the following:

- An extension to OpenFlow PacketIns allowing for context to be transmitted in-band
- A Windows OpenFlow Agent, which captures incoming and outgoing packets flows, sends them to the controller, and re-injects approved packet flows
- A Windows OpenFlow datapath prototype, which provides the functionality that enables the OpenFlow Agent, and proposal for future work
- A Windows Host context extraction, which extracts relevant context to be appended to PacketIns

WinSight's context extraction is powerful because it allows the monitor to consider the environment from which the traffic originated. This includes the current username, process ID, path to executable and executable name, and the window title. Including this data when monitoring a network greatly increases the scope of the network's security, as it allows network monitors to infer the intent of the application sending the packet. Host context data can be used to analyze traffic more deeply and automatically determine if the traffic is malicious. For instance, the network monitor can have a list of usernames who are allowed to send and receive HTTP packets. If a user not on that list, such as a guest or back-door user, attempts to send HTTP traffic, that flow can be blocked.

WinSight has a few key advantages over existing network security systems in addition to host context data. Unlike a standard firewall which operates only on traffic going to and from the Internet, WinSight operates on all traffic being sent within the network. This allows internal threats, such as compromised computers connected to the network, to be addressed. Additionally, WinSight does not require the computers to enforce security policy. In a traditional distributed firewall, when there is an update to security policy, that update must be pushed to the computers in the network. If the network is under constant threat, this makes new attacks difficult to react to. With WinSight, security policy is enforced by the OpenFlow controller, meaning security policy updates need only be installed on a single machine. This allows the monitor to immediately react to new attacks, and avoids the need to constantly push security policy updates.

To provide this protection, WinSight makes a few assumptions. It assumes the trusted computing base includes the operating system and all subsystems it is running on top of; this means that we do not protect against attacks that compromise the operating system, such as rootkits. WinSight also assumes that the OpenFlow controller is secure. Due to the nature of SDN, WinSight introduces a DoS vector: if the controller becomes unavailable, the host machines will be unable to forward packet flows. This is an inherent limitation of centralized software-defined networking controllers and is an active area of research for SDN [28].

In this paper, we discuss the following. Section 2 provides background information and research previously done in the field of network security which relates to WinSight, including an overview of software-defined networking and it's uses for security. Section 3 details the inner workings and design rational of WinSight. Sections 4 and 5 detail the performance and usability testing conducted on WinSight, as well as the outcomes. Section 6 details our final thoughts of WinSight, potential areas of improvement, and a trial deployment. While WinSight is designed to expand security policy design, we do not discuss policy design in this paper.

2 Background and Related Work

A network allows computers to transmit data packets to other machines that are connected. A local area network (LAN) is a type of network which is confined to a small area, such as an enterprise or home. Many LANs are connected to the Internet, and serve as a medium for computers to access Internet sites and services. An enterprise network is a type or LAN which is used by a business to connect its various departments. Enterprise networks allow businesses to efficiently share resources across departments, and often employ special security to control the network privileges of users and defend the network from malware.

2.1 Malicious Software (Malware)

Malware describes any unwanted program which engages in malicious activity on a computer. The impact of malware can have a variety effects, including unwanted pop-ups, data leakage, engaging in denial of service (DoS) attacks, and creation of back-door entrances to otherwise secure software. Some forms of malware are designed to propagate. For instance, a virus is a form of malware which inserts copies of itself into another program, becoming part of it. As the program is used, the virus spreads itself to other programs and machines, leaving a wake of infection. A worm is a form of malware which spreads across networks by exploiting vulnerabilities in target systems. Worms use file-transport features in target systems to spread autonomously, limiting the need for assistance from its creator or its victims. A trojan is a form of malware that masquerades as a legitimate piece of software. They are not inherently self-replicating, but can be sent across networks by malicious or naive users and other forms of malware. Through malware, it is possible for a computer to become part of a botnet, a network of machines infected with malware that all report to a central machine. Botnets can be used for flood-based attacks or mass data collection [11].

If a computer storing sensitive data is compromised by some form of malware, the damages can be costly. According to a survey presented by Ponemon Institute, the average company losses attributed to cybercrime in the United States was roughly $15.42 billion as of August 2015 [1]. In a survey of types of cyber attacks in 2015, by Ponemon Institute, 100% of respondents reported attacks from viruses, worms, and trojans; 97% reported attacks from general malware; and 36% reported denial of service (DoS) attacks [2]. It was also estimated by the Identity Theft Research Center that roughly 781 million data breaches had occurred within the United States in 2015, with an estimated 169 million records exposed [3].

Further evidence of the impact of malware is described by Kuchan Lan [18], who ran an experiment to describe the impact of DDoS and Worm attacks on a network. To detect attacks, Kuchan Lan [18] captured packets and ran them through a detection script which flags packets if a large number of source IPs connect to the same destination IP in a second. Once packets were flagged, they were manually analyzed to confirm the presence of an attack. Packet flow latency for flows less than 100KB and Domain Name System (DNS) look-up latency were measured to analyze the impact of an attack. Unsurprisingly, DDoS attacks were found to slow down both packet flow latency and DNS look-up time. Kuchan Lan [18] also describes worms to be able to infect many hosts in a short period of time. The Apache mod_ssl worm, also described as the Slapper Worm in the paper, infected hosts and can automatically trigger DDoS attacks. The Slapper Worm did not cause significant traffic by itself, but a coordinated attack from infected machines could cause a widespread DDoS.

In order to understand the dynamics of malware protection, Lelarge [20] models the risks of malware epidemics and the dynamics of overall security in a network. In the model, users who do not protect their computers by consistently updating their anti-virus software and operating system put a network at risk when connecting to it. The model assumes that infected computers can spread their infection due to the nature of the malware, and studies the epidemic spread of worms and viruses, as well as alerts and patches, in a network. In situations where all agents (owners of computers connected to a network) invest greatly in self protection, the general security of the network is higher. However, self-interested agents may not maintain their level of security as the overall security of the network is high enough that the cost of self-security is hard to justify. This leads to greedy agents discontinuing their protection to save costs, creating a downward spiral which results in an insecure network. In situations where protection is monopolized by a provider who is not responsible for the well-being of individual agents also results in an insecure network, as the provider has little incentive to provide quality protection. From this research, we can conclude that requiring agents to manage their own security in an enterprise network is not optimal.

2.2 Trusted Computing Base

Developing software to combat malware can be tricky, so developers must know the attack vectors they intend to close. This means we must define a trusted computing base. The trusted computing base (TCB) is the set of hardware and software which we must implicitly trust, whether trustworthy or not, to make any progress in a security analysis of a system [19]. Without specifying a trusted computing base, it could always be argued that some portion, such as the operating system or worse the boot-time routine of the hardware, is vulnerable. However, advances in security-related computing have led to a secure bootstrap, allowing us to trust that the low level components of our system have not been tampered with [4]. This means we trust that our BIOS, kernel, and operating system have not been compromised. It does not mean, however, that we can trust that there are no vulnerabilities in any of these systems. When developing WinSight, we assume every subsystem running below our userspace application and kernel module is trusted.

2.3 Firewalls

One of the most common protections against malware is a firewall. Firewalls were proposed in various forms as a way to protect vulnerable systems from a malicious network. The "modern" incarnation of a firewall is a packet filtering firewall invisible to the user of a network. It detects malicious traffic based on sophisticated rules created from years of observations of exploits, and it prevents any activity matching those rules from entering. Many of these rules are in the form of an a network access control list (network ACL). They can also be extended to represent user permissions to certain objects within the network, in addition to basic Internet restrictions. Network

ACLs are able to explicitly block a user from accessing a certain resource on a network based on the networking information of their request, such as their IP address or port number.

A firewall is responsible for enforcing network security policy at the junction of two networks by controlling which traffic is allowed to pass through it. Policy is defined as the decision-making process behind a firewall. There are many kinds of firewalls. A traditional firewall is a system at the boundary between two networks which has some mechanism to allow or disallow the passage of traffic [16]. A personal firewall is often used to enforce policy for a single machine. When describing the security base of a traditional firewall, we can imagine that the host is the single node interior of the network and everything else the exterior. The assumption that all firewalls have to make is that traffic on the inside of a network is trusted, because without user specifications or previous configuration they have no basis upon which to discriminate. This does not necessarily always prove to be true, as anything from unaware and compromised users to malicious users might engage in attacks which the firewall cannot detect.

Firewalls became necessary in the late 1980s as Internet use became more widespread and grew as an attack target. For example, when the Morris worm was released [26], the United States government realized German spies were extracting information from computer systems, and security quickly began requiring further attention. Securing systems at the user level was the first step, but many of the systems in use had bugs and vulnerabilities that left them exposed [6].

2.4 Distributed Firewalls

In the case of traditional and personal firewalls, both have room for improvement. A traditional firewall provides no insight into network-wide traffic, and a personal firewall provides no centralized control. The lack of centralized control with personal firewalls can make the overall security state of the network ambiguous, as well as make network-wide security updates tedious. The middle ground between these two is a distributed firewall which combines centralized control with distributed enforcement [6]. A distributed firewall is not a physical box in the way that a traditional firewall might be, but rather a specification of policy for all users of a network. Various schemes have been proposed, but all distributed firewalls must consist of a way to express network policy, a mechanism for distributing policy to end hosts securely, and a mechanism for enforcing policy at end hosts. With all three specified, any trusted end-host will enforce policy as specified by network operators. A distributed firewall removes the assumption of restricted network topology, removes entirely the performance bottlenecks inherent in such a restricted topology, and ensures that all network interior communication follows a specified security policy.

Ioannidis et al. [17] implement a demonstration distributed firewall using their proposed policy language KeyNote. The authors specify the language and policy enforcement, even a signing scheme, but not the distribution of policy. Instead they state that it could be distributed using various boot-time techniques or file sharing services.

The advantage of distributed firewalls is that they do not depend on restricting network topology. The often-cited example is that policy can still apply to a telecommuting user, reducing risk that security vulnerabilities might be transmitted over a VPN. However, while distributed firewalls are effective at blocking known malicious traffic, they are not as effective at detecting new threats. Similarly, a network ACL is only effective at blocking traffic based on defined rules, and can only block based on networking information. A traditional distributed firewalls and network ACLs are unable to see the context from which the packet was sent. Such context can include the current user logged into the source computer, the executable file name of the program sending the traffic, and even the window title of said program. If the context were included in the evaluation space, network administrators could have the ability to prevent whole programs or users from sending traffic, rather than just IP addresses and port numbers.

2.5 Software-defined Networking

A rising recent method of network management that is somewhat similar to distributed firewalls is software-defined networking (SDN). SDN separates two planes of operation in networking which are normally coupled together: the control plane and the data plane. The control plane describes the functions that dictate packets' immediate destinations in order to reach their final destination. The data plane describes the functions that dictate the method in which packets are forwarded to their immediate destinations. With SDN, the control plane is handled by an external SDN controller and the data plane is handled by the network switches. This allows network traffic control to be accessed from a single device, making it programmable. When applied to network security, SDN can be used to block malicious traffic from traversing the network. However, many forms of SDN for security are limited to only networking information, such as the source and destination IP address of a packet, when evaluating traffic. A policy that is set on a specific set of addresses is usually referred to as a fine-grained policy, whereas a coarse-grained policy will operate on larger groupings such as subnets or an entire protocol.

SDNs incorporate these ideas in varying ways, and one of the most common SDN standards is OpenFlow. OpenFlow is described as the first standard communications interface defined between the control and forwarding layers of an SDN architecture. It allows networks to be highly programmable and flexible by allowing remote management of a switch's packet forwarding tables. As its name suggests, OpenFlow is freely distributed and open source.

OpenFlow was built off of Ethane, presented by Casado et al. [10], which was a network configuration that implemented a central controller that enforces a global network policy on all network traffic. In order to give the controller full visibility, special switches are placed in the network which send all received packet flows, sequences of packets with the same source and destination, to the controller for evaluation. These switches can be implemented without any host modification and can be installed with minimal network interruption. Ethane offers the benefit of tracking the entire network topology on a single machine, which makes network state changes easier to account for. Ethane also enables a way to reconstruct all network activity, which is useful for diagnosing configuration faults and discovering unwanted activity.

A year later, McKeown et al. [22] presented OpenFlow, a standard which expands on Ethane to allow network protocol experimentation without exposing the internal workings of the switches. Similar to Ethane, OpenFlow works by having special switches placed in the network, which communicate traffic information to a controller. OpenFlow switches contain a packet flow table, and an action associated with each flow.

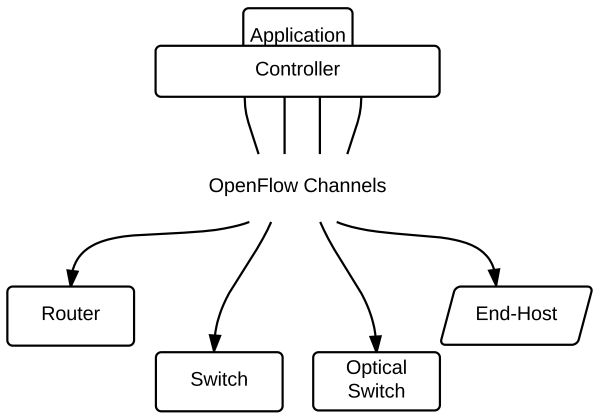


Figure 1: Heterogeneous network devices controlled by standardized OpenFlow protocol

Each unmatched packet flow is escalated to the controller via an OpenFlow header specified as a PacketIn message. The controller then makes a decision on the flow and relays it to the switch using a PacketOut, or in some cases a FlowModification (FlowMod). Based on the message from the controller, the switch will take one of several actions on the flow. These actions include forwarding the packets along the network, and dropping them entirely. OpenFlow specifies a standardized interface for flow matching and control protocols, as depicted in Figure 1, preventing network admins from having to write their own control protocols, and making the entire network programmable from the controller. This makes it possible for network administrators to write and implement network applications. These applications can perform actions such as granting special privileges to certain machines and blocking packet flows with specific values in their headers.

2.6 Software-defined Networking for Security Applications

Software-defined Networking presents a unique opportunity for security applications. It combines the policy enforcement power of middleboxes and firewalls as described above, with the lack of topology restrictions of a distributed firewall. This is all enabled by a standardized programmatic interface, allowing for arbitrary applications to be developed and possibly combined in novel ways [23]. This standardized interfaces makes applications broadly applicable to networks; they need not be aware of implementation details at the network layer. Applications are completely orthogonal to the network itself, allowing both to evolve independently.

Given this confluence of factors, many different security applications have been explored using SDN for enforcement. Firewalls have of course been re-implemented and distributed through the network [14] with later extensions to handle the statefulness of transport protocols [12]. Anomaly detection, normally a very expensive operation, was made computationally cheap by using OpenFlow PacketIn messages [23]. In a similar vein, OpenFlow made it simpler for a network to detect and mitigate a DDoS [9], though caveats about saturating a network's uplink still apply.

An interesting concept has recently been explored where policy resolution and enforcement are decoupled [7]. If an OpenFlow controller performs computationally expensive policy resolution calculations we might quickly overwhelm it, especially as it continues its role in policy enforcement. Instead we might be able to forward PacketIns or even redirect flows through "traditional" middlebox network appliances. The middleboxes can then perform some computationally expensive policy resolutions and return a response to the OpenFlow controller, which enforces the policy across the network in the form of flow table entries. In this way the negative side effects of middleboxes can be avoided. Only traffic which needs policy resolution traverses the middlebox, avoiding sub-optimal routing, and similarly the middlebox is not overwhelmed trying to process unrelated traffic.

2.7 Security of Software-defined Networking

Software-defined Networking provides new and exciting opportunities, for both network operators and attackers. The network operator has new found power and control over their network, but so too does the would-be network intruder [29]. SDN suffers from some of the same pitfalls as do traditional networks, and introduces several new ones.

The switch and data plane open up new vulnerabilities. An OpenFlow switch has limited resources for storing flow table entries, so an attacker could wreak havoc by generating traffic which causes the flow table to become filled. With a full flow table the switch now has to make a full PacketIn round trip for each new flow and network performance deteriorates significantly [13].

Attackers can also target the OpenFlow controller. If not properly hardened, an attacker could attack the host operating system in any number of ways and disrupt network availability. An attacker can impersonate the OpenFlow controller using a simple man-in-the-middle attack if the OpenFlow channel is left unencrypted. Worse yet, it is possible for attackers to infiltrate the OpenFlow controller and silently create fraudulent flow table entries. With such control, the attacker will have great power over the network [15].

2.8 Host Context and Security

Network security applications can leverage user-generated context to inform policy decisions.

Shirley and Evans [30] used context to determine whether traffic originated from a user action or not. In the latter case, attempts to access a previously-unknown address were manually validated by users with a modal dialogue. The authors use data gathered from a university campus to show that most users have relatively regular Internet usage habits. Accordingly, this approach would incur only a small initial workload and an even smaller ongoing overhead for the user. The authors described that future work might collect and aggregate acceptance and refusal from users to make decisions on a network rather than per user level. However, they provide no mechanism for collection of this information nor enforcement of network policy.

MacFarland and Shue [21] explore a slightly different take on context for security, allowing information extracted from a graphical user interface to highlight periods when “sensitive” activity is taking place on a machine. With this information system logs can be scanned during sensitive activity to identify malicious behavior. This same technology has been demonstrated on Windows in [32] using a tool called `Detours` which allows system calls to be replaced by arbitrary functions. This was used to implement an interceptor which pulled text from the GUI into logs for analysis. Both of these technologies can be used to enable real-time context extraction without user intervention.

Taylor et al. [31] develop a system that is the inspiration for our own. It allows

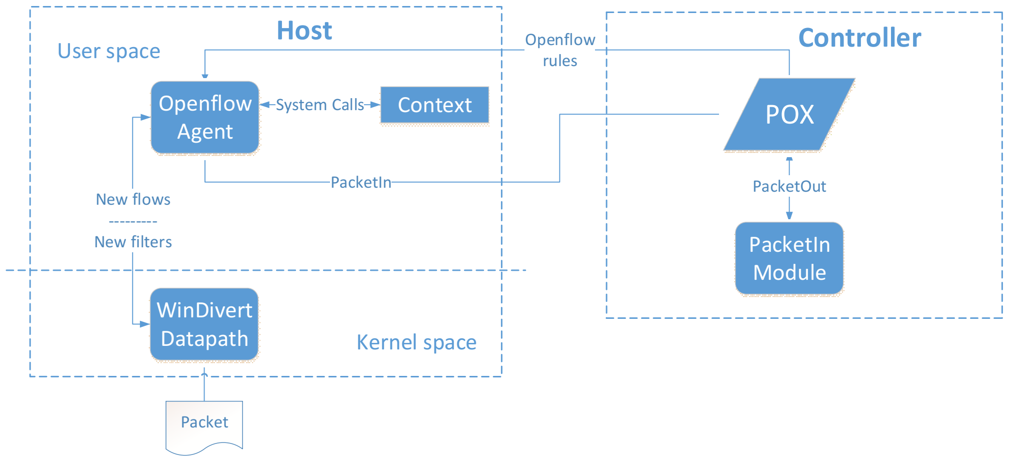


Figure 2: High Level Diagram

centralized control of network policy with fast updates similar to OpenFlow. Unlike the original conception of OpenFlow, it uses host machines for enforcement of policy rather than commodity Ethernet switches. This has the benefit that there is no possible way for the network to become saturated with rules. If a host wanted to perform communication with some other host, it needs at least the memory available for a filter rule, and with memory capacity growing so quickly the point is nearly moot. This implementation was designed for Linux and leans heavily on the Linux kernel subsystem `netfilter chains` which are optimized for high performance. Unfortunately, the original prototype was plagued with connection setup latency because the agent that installed filter rules was written in Python and ran slowly. Additionally the implementation was not compliant with OpenFlow, making integration into existing networks difficult at best.

3 System Architecture

In this chapter, we introduce the design of our OpenFlow-compatible and context-aware distributed firewall, WinSight, and the details of its implementation on Windows 7+. Further, the libraries we use claim to be compatible with Windows Vista.

3.1 Overview

Our design consists of three main components: WinDivert, a kernel space packet filter for intercepting and reinjecting network traffic; a hand-written OpenFlow Agent, interfacing with WinDivert and extracting contextual information; and POX, a widely-used OpenFlow controller. The kernel filter catches all the traffic sent to or from the host machine that is not destined for or sent from the controller. The OpenFlow agent gets packets from the kernel datapath and checks for an existing matching rule. If a rule is not present, the agent gathers context for the packet then escalates it to our POX module. When the agent receives a response from POX, the packet is either dropped or re-injected depending on the decision. POX is an OpenFlow controller written in Python which makes decisions about whether to allow or reject a network flow. We created a module for POX to extract our context information and make decisions based on it.

Our project targets Windows 7, but should be backwards-compatible to Windows Vista and forwards-compatible to Windows 8 or 10 as well. This assumption is untested, but is based on information reported by the documentation for the Microsoft Developers Network and external libraries. Since WinSight is targeted towards corporations and large businesses, it is in our interests to develop primarily for Windows 7 as it is currently the most common OS used. However, support for Windows 10 is desirable given Microsoft is encouraging businesses to update to it as the new standard, and it may be widely adopted in the coming years.

The libraries used are exclusively native Windows APIs defined as having long-term support, including the WinSock and Windows Driver Framework libraries. Some libraries are included for function calls when providing context, such as `libpsapi` and `libiphlpapi`. These libraries also are cited as being both backwards- and forwards-compatible with other versions of the Windows operating system. We are using MingW32 for compilation of the Win32API and WinSock code [25]. Next we examine the various pieces of our system architecture.

3.1.1 OpenFlow Agent

The OpenFlow agent is the heart of WinSight. The agent enforces network policy by inspecting all traffic, escalating unidentified flows to a central controller, and gathering contextual information from the operating system. Network traffic flows in and out of the agent through WinDivert, a kernel driver that interfaces with the Windows Filtering Platform. When a packet comes in, the agent checks it against our existing flow tables; if it matches the agent passes it along, and if not the agent escalates it to a central controller. If there is no match in the flow table, an explicit `DROP` rule is added per the OpenFlow version 1.3 specification as an optimization. On escalation, the agent calls into a number of Windows APIs to gather information about the flow, which are discussed later, that describes what process originated the packet and who started that process.

When deciding whether to drop or re-inject a packet, our application inspects a flow table. This table is represented as a binary tree and stores an action (represented as an integer) and expiration time (a configurable `time_t` value, seconds since the Unix epoch) as its values. The table is indexed using an integer representation of the traditional network 5-tuple, consisting of the binary representation of the source and destination IP, the source and destination port, and the protocol. If the flow table does not contain an entry for the tuple generated from the packet headers, it will buffer the packet. Next, the agent sends an OpenFlow PacketIn message to the controller to determine if the flow should be allowed.

This PacketIn message contains a copy of the source packet and appended context data, formatted as described in subsubsection 3.2.2. If no context data could be found, a binary zero is appended. Prior to sending the PacketIn, the packet is appended to a ring buffer, allowing us to look up a packet by a continually increasing 32 bit buffer ID instead of requiring the controller to return the packet data.

We chose a circular buffer as opposed to a map because we consider the relative efficiency of O(n) versus O(1) a reasonable loss when compared to the memory overhead. This holds particularly because in the case of a "drop" decision, a PacketOut will not be sent, requiring manual tracking of old packets and removal of an old node from the map. However, a circular buffer will wrap around and overwrite an old value without

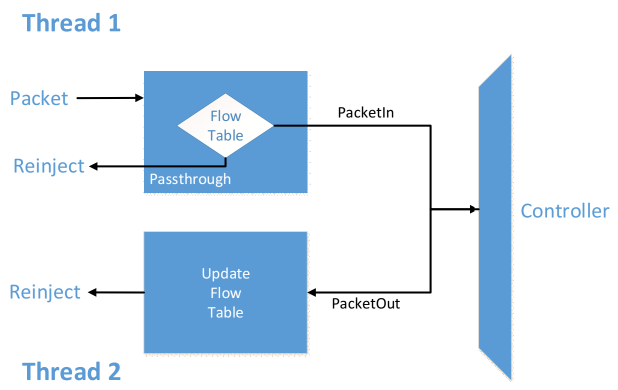


Figure 3: Thread Functionality

any additional statefulness. The ring buffer is indexed by a truncated ID, but also contains a non-truncated ID to allow verification that the packet was not overwritten by buffer overflowing between sending a PacketIn and receiving a PacketOut. If the full ID between the PacketOut and the entry in the buffer do not match, the PacketOut is ignored as we have already dropped the old data. If a packet is dropped, it is left in the buffer until the ID sequence wraps around and it is overwritten with a new packet. The buffer is currently sized for 2048 packets, but this is trivially modifiable if in testing that is found to be insufficient.

To avoid blocking and thereby increase performance in our agent, we employ pairs of threads. The number of thread pairs is passed in as a command line argument. The first of each pair sets up a communication channel with the OpenFlow controller, spawns its partner thread which is responsible for re-injecting packets, and then blocks on receiving a packet from the WinDivert handle (which is thread-safe according to their documentation and our tests). When the first of the two threads receives data from the handle, it either immediately re-injects it or escalates the packet to the controller depending on the existence and value of a flow table entry. Since both threads share a socket connected to the controller, the re-injection thread is able to block on a `WinDivertRecv` call without impeding the primary thread, which is responsible for receiving packets from the network stack. When the second thread receives a PacketOut from the controller, it then checks the packet for a buffer ID and re-injects the appropriate packet, if the buffer IDs are an exact match. It then adds an entry to the flow table indicating the appropriate action. The flow table that the thread pairs use is shared globally, but the packet buffer is only shared between the two threads in the pair. This optimizes the program such that there are not duplicate PacketIns for the same flow and each set of threads can buffer their own packets, minimizing buffer overrun.

3.2 WinDivert

Windows Packet Divert [5], or WinDivert, is a kernel mode driver that interfaces with the Windows Filtering Platform to divert traffic out of the kernel and into userspace. During initialization, WinDivert creates a handle into the Windows Filtering Platform [24] and registers callouts for the transport layer. When the Windows Filtering Platform kernel-mode filtering engine receives traffic it calls out to the WinDivert code. This code classifies the traffic according to a user-specified filter, detaining any that match. After blocking the specified traffic, WinDivert places it in a queue to be read by a userspace client. WinDivert also exposes the Windows Filtering Platform injection APIs, thereby allowing userspace code to inject packets into the network stack. Communication with WinDivert happens through a Windows file; all interaction with WinDivert from userspace is actually through IO controls (ioctls). WinSight uses WinDivert to pull all traffic into userspace where it can process it and reinject compliant traffic only.

From userspace we initialize WinDivert by calling `WinDivertOpen` with a filter. The WinDivert driver gets installed and runs through its initialization routines. WinDivert sets up a Windows Driver Framework I/O device, opens a Windows Filtering Platform handle with `FwpmEngineOpen0`, creates a Windows Filtering Platform injection handle with `FwpsInjectionHandleCreate0`, and installs its callouts into the Windows Filtering Platform. There are two injection handles created, one for IPv4 and another for IPv6. A total of six callouts are installed: three each for IPv4 and IPv6 in the inbound, outbound, and forward layers.

The Windows Filtering Platform kernel-mode filtering engine consists of the following elements: shims, the filtering engine, layers, sublayers, and callouts. Shims are closed systems provided by Microsoft that extract information for each layer of the filtering engine, which consists of multiple layers that correspond roughly to the OSI layered-network model. Each layer consists of sublayers, where filters and callouts can be installed. Finally, callouts are independently-developed code that performs some kind of packet analysis not possible with the filtering engine, such as WinDivert.

A filter or a callout can result in one of four actions: `CONTINUE`, `BLOCK`, `ACCEPT`, or `VETO`. A filter will statically match against packets, and if it matches some action is applied, otherwise it will return `CONTINUE`.

`CONTINUE` is a no-op. The MSDN documentation does not include `FWP_ACTION_CONTINUE`, but it does exist in the APIs. `BLOCK` and `ACCEPT` do as their names suggest. `VETO` is used for blocking even a higher-priority filter, which is critical for an application such as ours.

Within a layer in the filtering engine, many filters could come to different decisions about a packet. There are filter arbitration rules to handle this. Each sublayer is a set of filters which are ordered by priority. These filters are executed until one of them matches and returns an action other than `CONTINUE`. All sublayers are executed for a given layer, and higher-priority sublayers take precedence. If two sublayers return conflicting actions, arbitration determines which sublayer has higher priority. WinDivert installs its callouts into a separate high-priority sublayer in six different layers. It does this to avoid ever being preempted or skipped by some other filter, and to ensure that its decision always takes precedence.

When a packet reaches the Windows Filtering Platform network layer it calls out to `windivert_classify_callout`. It first checks to make sure that a packet was not injected by itself. Without this verification, then a packet would be filtered, reinjected, and filtered again endlessly in an infinite loop. Next it calls `windivert_filter` to check the packet against an internal filter generated by parsing a string provided by the user. WinSight wants all traffic in the network, so it uses the filter `"true"`, excluding the traffic sent to or from the controller. If the packet does not match the filter, it is immediately reinjected into the network without ever leaving kernel space, or copying the packet inside of WinDivert. If it does match the filter, then the packet gets blocked and copied into a reader queue awaiting a user application, and all pending applications are notified.

The read queue is a standard I/O queue from the Windows Driver Framework with manual dispatching. Manual dispatching means that WinDivert maintains full control over when read requests are serviced. WinDivert needs to maintain control because network activity is unpredictable and, once it accepts a read request, it either has to service it or cancel it; therefore if it blocks waiting for a packet, the kernel blocks with it. To avoid that, WinDivert keeps user applications blocked until it has a packet to deliver. Read requests are serviced by WDF work items, which are basically kernel-mode worker pools. A work item services a request by pulling a copied packet out of the head of the read queue and copying it into a buffer in user space, using standard WDF APIs.

As with all queues in networking applications, this read queue can become filled; WinDivert has two strategies for handling this. The queue has a maximum size of 1024 entries: and if a packet arrives and exceeds this length, it gets dropped. In addition WinDivert has a timer which manages a Boolean `ticktock` and purges from the read queue any packets that do not match. By default, the time runs with a period of 512 ms. WinDivert drops from the tail in a fairly conventional way, and then purges the head of the queue periodically to keep the queue from getting overly stale.

From userspace, WinDivert provides a DLL that includes a simple API for clients to use. `WinDivertRead` handles creating an IOCTL representing a read request and submitting it to the file object that WinDivert exposes. `WinDivertWrite` similarly handles a write to the WinDivert device and initiates an injection into the Windows Filtering Platform.

The write side of WinDivert is more straightforward. A user-space application issues an IOCTL write request and WinDivert immediately serves it by calling out into one of three Windows Filtering Platform functions depending on the destination of the packet: `FwpsInjectForwardAsync0`, `FwpsInjectNetworkSendAsync0`, or `FwpsInjectNetworkReceiveAsync0`. These are all asynchronous calls which return immediately, before the injection operation has actually completed and the packet data is no longer needed. Instead, WinDivert registers a callback with these functions and when injection is completed, it frees the memory used for storing the packet and completes the write request, unblocking the calling application.

We do not fully understand WinDivert's decision to block an application until the injection completes. It is logical for an application that wants to be sure that the injection succeeds, but it does not match the needs of WinSight. WinDivert has to copy the packet out of userspace to submit it to the Windows Filtering Platform, so the application does not have to be blocked for memory safety. WinSight does not care if injection fails, and if reinjection fails WinDivert could indicate that a network buffer is full. In that case, the traffic should just get dropped, as is standard in most network appliances, thereby eliminating the need for blocking behavior.

3.2.1 OpenFlow Controller

We use POX to act as our OpenFlow controller. We selected POX due to the Python development environment and ease of extension. POX has an event-driven architecture, allowing us to hook directly into the PacketIn event to parse the additional context data our system sends. The work done on POX was trivial: we simply installed a current version and added a module capable of understanding the additional context we were appending to the packet.

This module inspects all PacketIns for context data and makes some policy decision based on it using hard-coded logic, as we simply needed to test that what was being sent was correct and actionable. If the PacketIn describes a flow that complies with our basic policy expressions, the module generates a PacketOut message corresponding to the flow and communicates it to our OpenFlow Agent. If the flow does not comply then we send nothing and exit the handling function; the flow will be silently dropped by the host OpenFlow agent due to the default `DROP` rule that is installed automatically.

3.2.2 Context

For the scope of this project, we define context as additional information associated with a flow and its resultant PacketIn, which allow a controller to make better-informed policy decisions. There is tremendous value in this data, but collecting it is nontrivial and it complicates the packet escalation process. Context can only be extracted in a user-space scope where we have access to local processes, their security descriptors, and other similar information. This means that there is some necessary computation and delay getting information in and out of kernel space. Once this data is collected we have to send it along to our central controller. OpenFlow provides no standardized mechanism to do this.

There are several options for sending additional contextual information associated

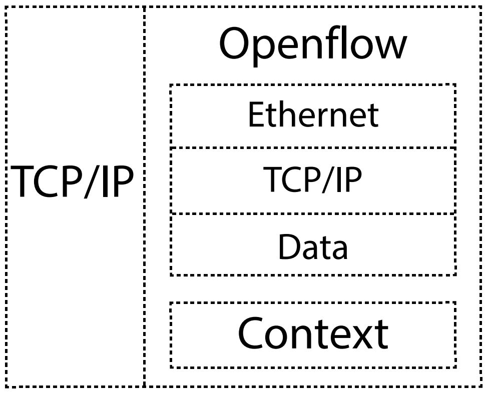


Figure 4: Context within OpenFlow PacketIn message

with a PacketIn. Our OpenFlow agent could send a PacketIn, and our controller could query the agent for information it desires. This approach has the benefit of saving some communication cost if context information is discarded, but forces the controller to have deep buffers for PacketIns while it waits for context query responses. To mitigate this controller-side buffering expense, we could instead send the context information appended to our PacketIn. This forces the buffering burden on the agent, which prevents us from easily overloading a single point of failure. The cost to this approach is we have to decide how we are going to append this information.

One option is to append the context information to the TCP stream, *outside* of the OpenFlow PacketIn message envelope. If we do this, then we have to strip this information out of the TCP stream, before it reaches the OpenFlow controller. This is problematic as our implementation would require either modifying the TCP stream or performing a man-in-the-middle attack our OpenFlow channel to insert and strip this data. If we directly modify the TCP stream, we would need to write low-level code for the kernel of our controller host to transparently strip the information and deliver it to the controller over another path. This couples our deployment to a particular operating system and also violates the semantics of TCP streams. We could instead man-in-the-middle attack our OpenFlow channel and have a userspace context stripper, which decouples us from an operating system, but requires us to establish some other secondary channel to the OpenFlow controller.

The other option for appending context is to insert it *inside* the message envelope. The OpenFlow specification states that the data portion of a PacketIn is meant to contain a partial Ethernet frame, but we found no part of our system that actually enforces this assumption. We can then just insert whatever data we want into the data field, including our context. In our demonstration, we send entire packets in our PacketIns, meaning that our controller can do full deep packet inspection if it wants to, and the context data will not interfere, as it is distinguishable from the rest of the packet due to the included size. We acknowledge that the ability to arbitrarily append data to an OpenFlow PacketIn is a viable attack vector, as there is no reason malicious data could not be inserted in the position of our context. However since we assume that the subsystems are trusted, no other program with access to the packets should be able to intercept and modify them maliciously.

In appending data to PacketIns, we used a hard-coded set of fields. The data constituting the context is newline-delimited, and should a newline be present in any of the context data, it is replaced by a space. We extract context data by finding the process originating a flow, and using that knowledge to look at the source and destination ports and matching them to a pairing in the TCP or UDP state tables. We can then extract the following:

- `[int]` Process ID
- `[char, <256 bytes]` Path to executable, including executable name
- `[char, <256 bytes each]` User and domain the process is running under
- `[char, <256 bytes]` Service name if associated
- `[char, <256 bytes]` Window title, if the process created one
- `[int]` Length of above information, or binary zero if none found

Together, this information tells us about what the process is, what it is doing, and who is running it. One particularly interesting field is the "window title". The content of the field is often not particularly useful, but its absence indicates a background process of some kind. If not also associated with a service, this sort of application invites suspicion and should be scrutinized closely.

4 Experimental Design

To test WinSight on Windows, we devise a number of methods to both explore its functionality and the various vagaries of Windows. We run a series of tests which attempt to block network connections from being setup to verify the capabilities of WinSight. We also characterize the performance of WinSight, in particular its effect on connection setup latency and overall connection throughput. Finally, we gather traces from the controller while a user engages in typical office environment tasks: browsing the web, reading email, and editing documents in a word processor and spreadsheet tool. With these traces we characterize the behavior of Windows to inform policy decisions made with context information.

All of these tests were run within VirtualBox using a Intel i7 quad-core 4.0 GHz CPU, of which the virtual machines were given two logical cores and 4 GB of RAM.

4.1 Blocking Tests

We performed numerous blocking tests, which can be broken down into the two broad groupings of *categorical* and *content*. Categorical blocks refer to blocks made on a field of the packet header and context tuple, such as the username or application path. A content block, on the other hand, is performed against the content of a packet as seen by the controller. This kind of block typically would be performed using deep packet inspection, but can also be used in the PacketIn handler in the OpenFlow controller.

We perform the following types of categorical block:

Application A block against the end of the executable path in context, blocks against the name of an application

User A block against an NT user name or domain

Window Title A block against the current window title of an application

Each of these blocks is accomplished with a simple conditional in our OpenFlow controller logic. If WinSight matches one of these fields, it does not return a PacketOut or FlowMod. Otherwise, the controller returns a PacketOut or FlowMod and communication continues normally.

We can additionally block against any of the traditional packet header tuple fields, but this is trivial in OpenFlow, needing only a flow table modification message. These tests exercise the capability of the host agent extracting context and the controller properly reading it.

We perform a single proof of concept content block. In this test we block any datagram which contains the string “virus”. This allows us to test our system’s ability to perform a deep packet inspection block. To ensure that our OpenFlow controller sees the packet we are looking for we use the utility `netcat` over the UDP protocol. In this way, we ensure that the first packet sent over a flow is a datagram containing the string virus, as opposed to something like an empty SYN packet.

4.2 Performance Tests

To characterize the performance impact of WinSight, we run connection setup latency and throughput tests.

The connection setup latency tests are evaluated by browsing to various benchmark websites. Using the *Google Chrome* web browser, we are able to get a trace of network activity. From this we get an understanding of the total time spent loading a page, a correlate for throughput and user experience, and for each connection made the Time To First Byte (TTFB). TTFB lets us see the impact that WinSight has on TCP stream setup.

We tested against the following websites:

example.com a very simple website with only a single HTTP GET request to load

cnn.com a complex website, consisting of many multimedia components all served from various Content Distribution Networks

wpi.edu a typical Content Management System website with mostly local content

nasa.gov another website with mostly local content but lots of multimedia assets

The distinction between local and non-local assets is important to WinSight, as with non-local assets the packet header will change (different IP address) and the system will have to negotiate a new flow. While `wpi.edu` will have only a few different packet header tuples, `cnn.com` might have many and require significantly more flow setup and will take longer to load.

In addition to these tests on HTTP performance, we also run an ICMP ping latency test. For all tests, we run the test both with and without WinSight to see the impact that it has. The test is duplicated to illustrate the overhead required by PacketIns versus the latency introduced by the agent itself.

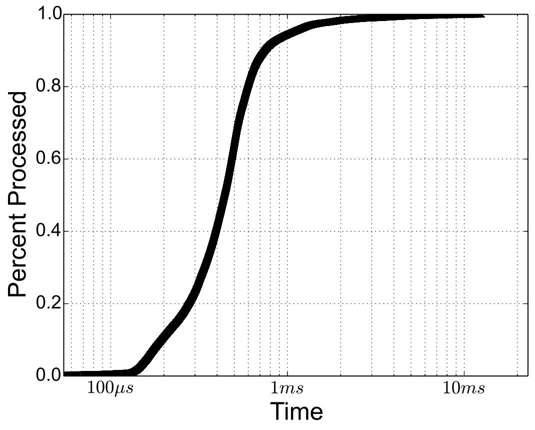
4.3 Agent Profiling

To test the internal performance of our application, we used the `QueryPerformance-Counter` function set. These functions provide an API that does not suffer from processor affinity or rely on `RDTSC`, which may be skewed. To ensure that the readings are not affected by processor frequency, power optimizations are disabled and minimum processor utilization is set to 100%. Overhead between measurements is taken to ensure that the times represented are accurate depictions of the calls they encapsulated and do not count the time required by the performance counter function calls. Due to the volume of data, measurements are logged to files such that every section of the code, excepting code responsible for behavior including variable initialization or memory allocation, is included.

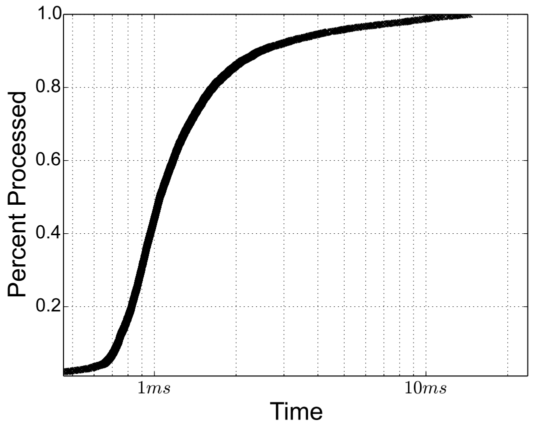
5 Results and Discussion

We present and discuss the results of our experiments on WinSight's performance and functionality. The graphs above were generated from multiple trial runs, so any discrepancies in the counts of data are due to updated metrics. We discarded any outliers beyond 3 standard deviations, the count of which are listed below their respective graphs.

WinSight performed well, with a few performance impacts. We were able to block traffic based on several context matches, including the process, path to executable, and service name. In terms of performance, The first packet of each flow sent through WinSight is delayed significantly, but the sequential flow is virtually unhindered. There are also rare times when packet reinjection takes much longer than normal, a flaw which we attribute to the Windows Filtering Platform.

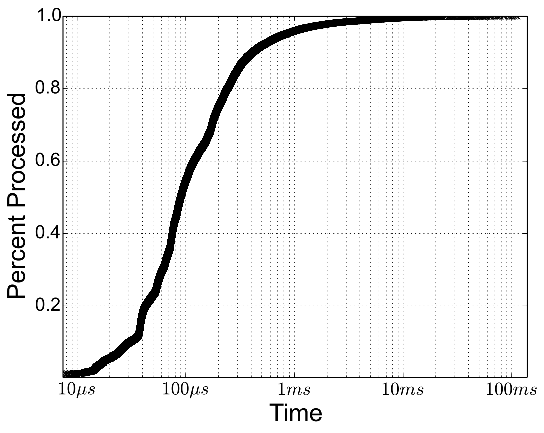


(a) Processing time for parsing packet headers, n=163,721 with 282 outliers. (0.172%) This includes network state table lookup resulting in the associated process ID.

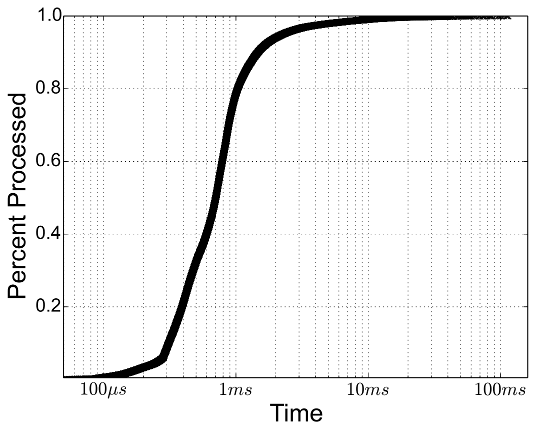


(b) Time required for packet context extraction, n=6,407 with 101 outliers (1.55%) This includes all context extraction based on process ID: username, executable path, service name, and first window title.

Figure 5: Context extraction steps, these take a mostly negligible but non-zero time.

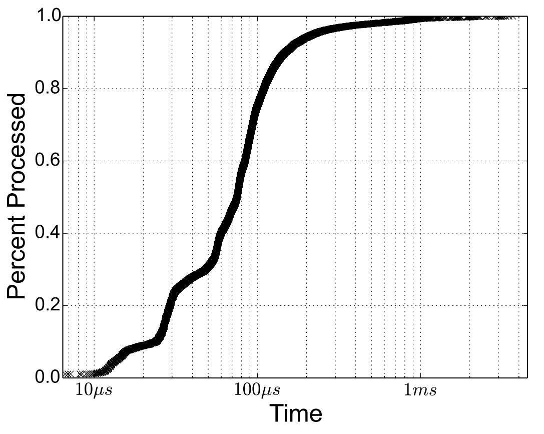


(a) Reinjection time for all packets including those not matching flow rules, n=81,696 with 64 outliers (0.08%)

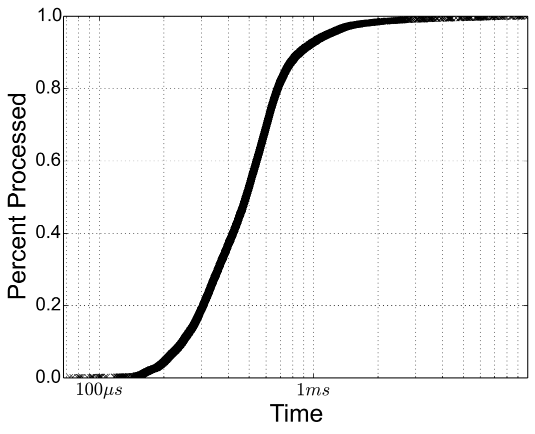


(b) Handling time only for packets matching flow rules, n=76,708 with 72 outliers (0.09%)

Figure 6: Time taken to reinject packets into the network with blocking functions. These functions block until packet is redelivered by network stack, and occasionally have very large latency.

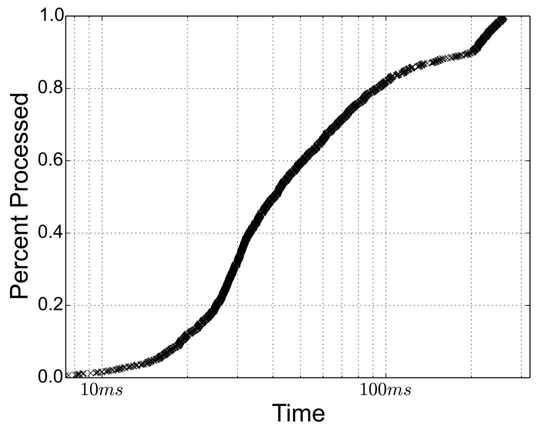


(a) Reinjection time for packet all packets including those not matching flow rules, n=18,826 with 49 outliers (0.26%)

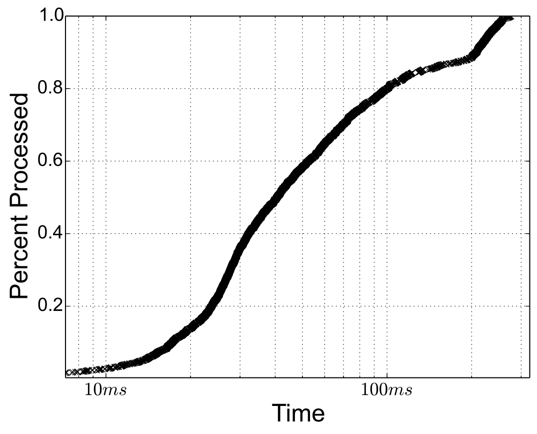


(b) Reinjection time only for packets matching flow rules, n=17,192 with 76 outliers (0.44%)

Figure 7: Time taken to reinject packets into the network with asynchronous functions. These functions do not block until packet is redelivered, so these results show that queuing a packet for redelivery is fast.



(a) Flow escalation end-to-end, from diversion to reinjection and including controller round trip, n=1,566 with 41 outliers (2.55%)



(b) Round Trip Time to controller, n=3,136 with 77 outliers (2.4%)

Figure 8: Comparison of total end to end time for an escalated flow and controller round trip times. Flow escalation is clearly dominated by the controller round trip, and not the reinjection.

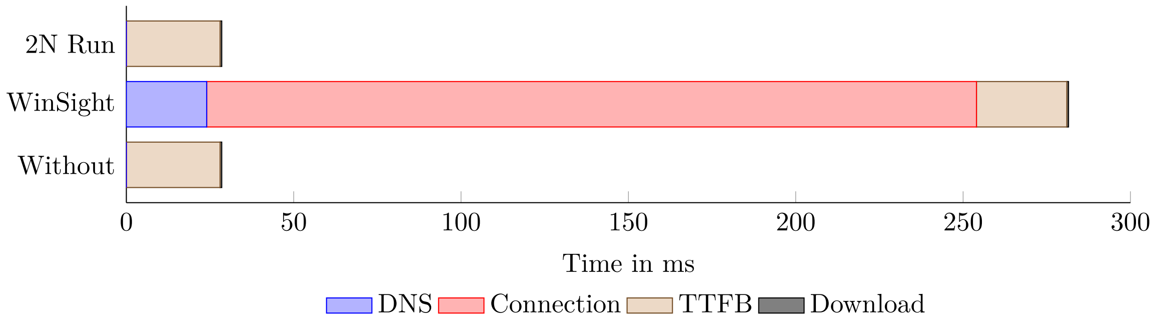


Figure 9: HTTP load times for example.com, blocking

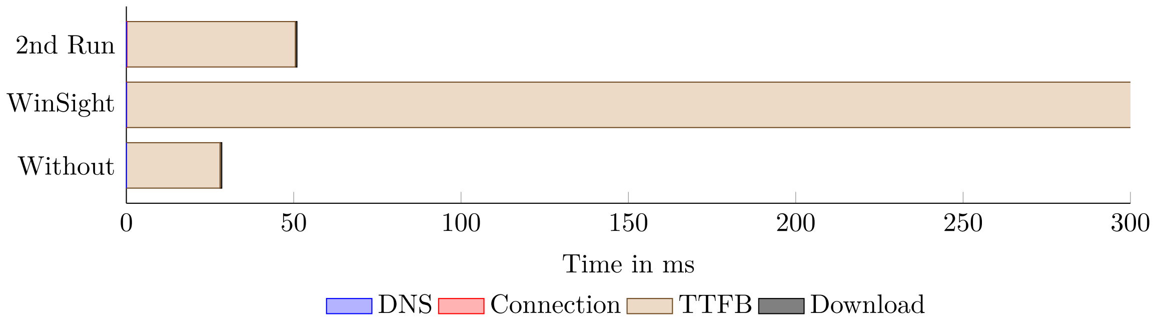


Figure 10: HTTP load times for example.com, asynchronous

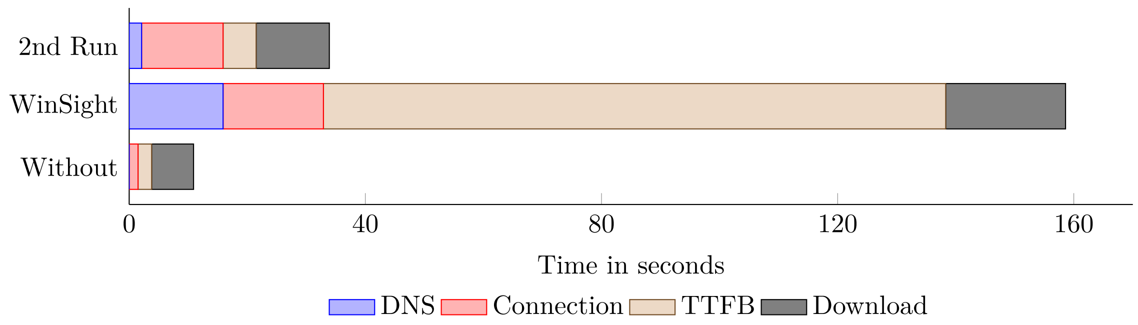


Figure 11: HTTP load times for cnn.com, blocking

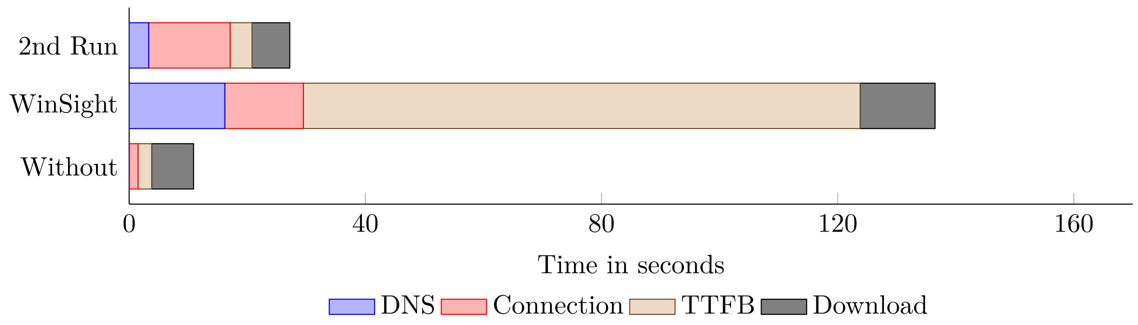


Figure 12: HTTP load times for cnn.com, asynchronous

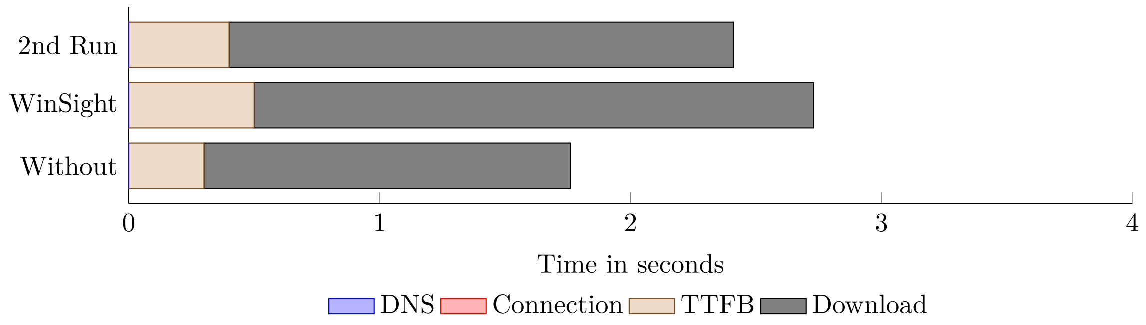


Figure 13: HTTP load times for wpi.edu, blocking

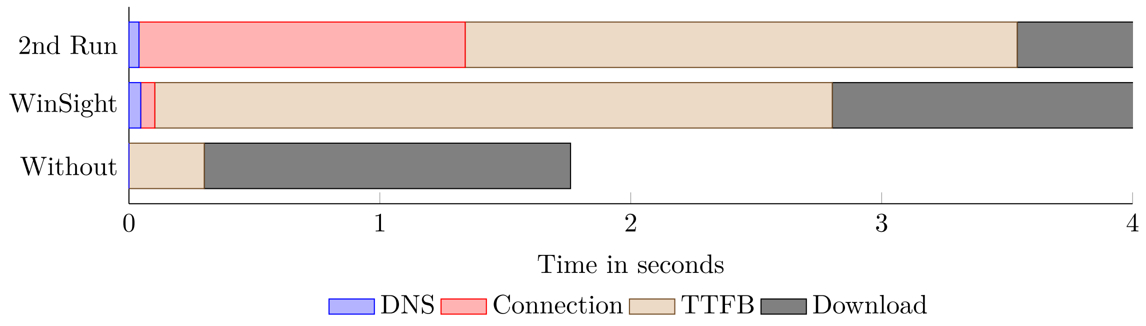


Figure 14: HTTP load times for wpi.edu, asynchronous

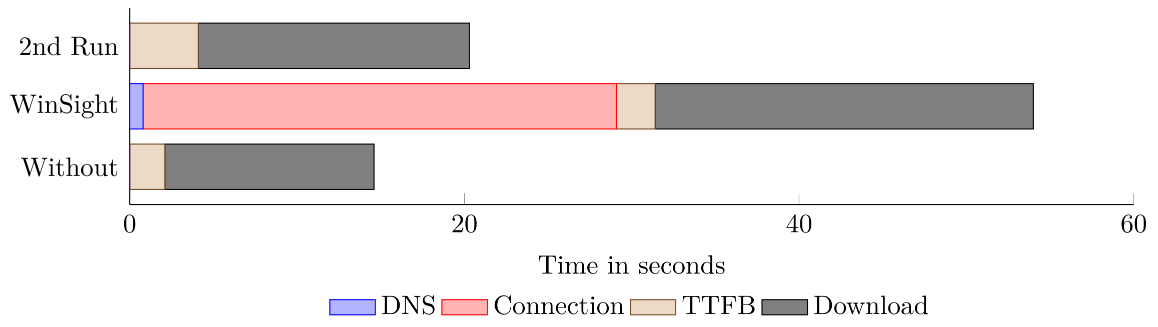


Figure 15: HTTP load times for nasa.gov, blocking

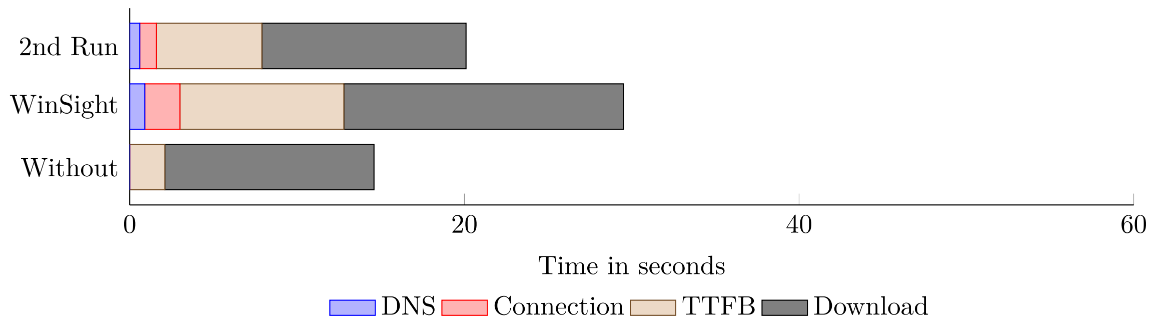


Figure 16: HTTP load times for nasa.gov, asynchronous

5.1 Blocks

At the highest level, we ran a series of tests expected to block various types of traffic. Our tests were able to block Internet traffic based off of the user or domain the process was running under, as well as a path to the executable. Partial matches on the window title were also found to be effective, but should not be relied upon as the sole basis of discrimination. Window titles are subject to complications such as non-standard titles between various applications and the ability for one process to spawn multiple windows. Additionally, the service name is included for the process, which allows detection of many malware processes which attempt to contact either a command node or download more malware. These could likely be identified by a network flow that does not have an associated service or window created, though it is of course possible for another legitimate process to have the same behavior. The final block was a UDP datagram with the content “virus”, showing that our module has access to the full packet contents.

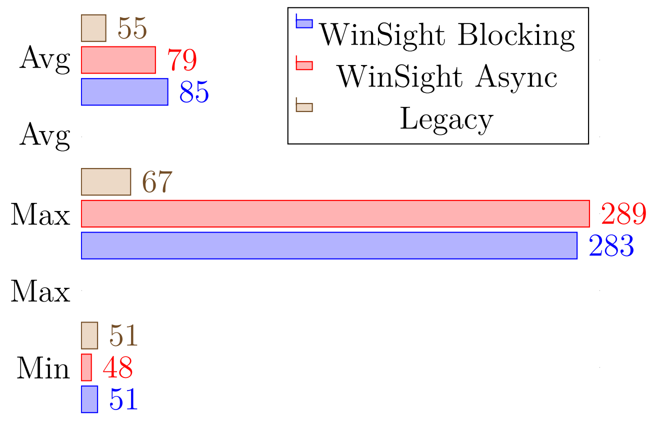


Figure 17: ICMP ping latencies reported by Windows ping.exe (in ms)

5.2 Performance

Figure 17 shows the results of the first test, a 10-packet ICMP ping to `google.com`. The times matched almost exactly, with the exception of the first ping on the WinSight client taking 283 milliseconds to perform the PacketIn/PacketOut communication. The numbers for asynchronous communication were very similar, with an initial connection of 280 milliseconds and a sequential average of 79 ms.

Figure 9 through Figure 16 show the results of our website load performance test. The webpage load times are scaled relatively, so on a page like `cnn.com` with a large number of video or image CDN requests, load times of something like a JavaScript file which takes less than a quarter or half a second are discounted, whereas on WPI's homepage, a few hundred milliseconds are significant. A final note is that due to Chrome's parallelization of network requests, the times shown here do not represent real-time, rather the time required to complete the given operation.

In general, the initial load times are significantly higher than the legacy load times, but this is to be expected due to the need to approve every flow connection. The average difference between a given file transfer on the first WinSight run vs. legacy was between 2 to 4 times longer, though some of them were significantly longer, as the graphs showed. A number of the requests would be blocked on one of the categories (usually connection initiation or content download) for between 5-10 seconds, but occasionally more. These delays were caused by packet reinjection and details are described in the next section. Typically, on the second run, the disparity in load times is greatly decreased to around

1.5 to 3 times longer. However, this is still rather high for a cached entry. After reading through the WinDivert source code, we found that they used a tick-tock queue purging algorithm, in which some amount of the queued packets (both inbound and outbound) are dropped every 512 milliseconds. Another possible reason for the high overhead is discussed later and deals with the reinjection of packets. These are issues that may potentially be solved if packets were not all buffered into userspace and the flow table was at least partially implemented in kernelspace, as it would bypass WinDivert.

5.3 OpenFlow Agent Performance

This section follows the path of a packet through our OpenFlow agent and performance bottlenecks along the way. A quick review of subsubsection 3.1.1, a packet follows these steps when passing through our OpenFlow Agent:

- A packet is received from the kernel through WinDivert
- The agent extracts header information and determines whether escalation is necessary
- If not, the packet is returned to kernel through WinDivert
- Otherwise, context extraction proceeds
- TCP/UDP state tables are acquired and interrogated for a process ID associated with a flow
- Other context is examined based on process ID
- The packet is stored in a circular buffer for later resending
- A PacketIn message is constructed and sent to the OpenFlow controller with context appended
- A PacketOut message *may* be received at a later time, at which point the packet is returned to the kernel through WinDivert.

Of these steps, the ones of most significance are header parsing, context extraction, and the reinjection function calls.

The expenses of header parsing are illustrated in Figure 5a. The median performance for this grouping of operations took 442 microseconds, and 95% fell under 1.08

milliseconds. However, the top 1% were above 3.57 milliseconds. The reason for the relatively expensive top percentage is the cost of translating a TCP/UDP port pairing to the owning process ID. The principal cost appears to be in the retrieval of the state table, and there was no discernible difference between the TCP and UDP protocols.

Figure 5b shows that when performing context extraction, half of the samples took under 1.06 milliseconds, and 95% below 5.65 milliseconds. The largest expenses of context extraction are split between username processing and associating the PID to a service.

As another part of the host agent, we also had to implement a flow table for OpenFlow. This implementation was efficient, with the tuple look-up taking negligible time, 99% of 81,800 measurements finishing in under 11 microseconds. All other operations performed similarly, so we do not believe this to be worthy of future analysis. The operations that are used to prepare for communication to and from the controller were found to be uninteresting. These steps include interacting with the buffer and creating/parsing OpenFlow messages.

After OpenFlow processing, whether escalated to the controller or passed back to the kernel, Figure 7a shows the time required to reinject a packet. As is visible, most packets can be reinjected quickly, with 95% of all reinjections taking less than 226 microseconds. Using the asynchronous method, the 99th percentile was 962 microseconds. However, synchronously, some reinjections took extremely long, into low single-digit seconds. These are not represented in the graphed data as this data was asynchronous and, even with synchronous calls, those data points are statistical outliers. Regardless, those spikes occurred approximately 0.1% of the time, which is significant in network traffic. We provide no complete explanation for why the code exhibits this behavior, and the asynchronous calls may simply move this delay out of the foreground and not solve the underlying issue. Our review on the code indicates that requests are blocked waiting on completion of a request inside of the Windows Filtering Platform, and that source code is not available.

Figure 7b illustrates the full time required to receive a packet, check the flow tables, and reinject the packet. Figure 8a shows the same situation, but also entailing a full round-trip to the controller. The network times are depicted in Figure 8b. The passthrough chart is relatively uninteresting, the 95th percentile shows that packets are reinjected within 1.21 milliseconds, which is within tolerances due to the slight inefficiencies mentioned previously. However, the round-trip times incurred are significantly more expensive. Note the approximate shape and order of magnitude of both graphs. At the 50th percentile, the round-trip time requires 41.73 milliseconds, and the full escalation time is 41.62 milliseconds. This suggests that packet escalation is dominated by the round trip communication with the controller. At the 95th percentile, the total time is 243.95 milliseconds versus a RTT of 273.8 milliseconds, which again suggests the round trip dominates and that it is more volatile. This is reasonable considering the use of a Python controller running inside a virtual machine.

Finally, we note that timing the receipt of a packet is difficult due to the blocking nature of the `WinDivertRecv` function and asynchronous calls to `WinDivertSendEx`. We cannot differentiate between either blocking on an empty queue and time spent copying a packet, or the time spent between dispatching a packet and its reinjection, without modifying WinDivert.

5.4 Discussion

Our results suggest that an approach using a divert socket into userspace like WinDivert has surprisingly good performance in the median case, and bad performance in the worst case. By far, our performance is dwarfed by expensive reinjection, which should be the target of future improvements. There are improvements to be made by using a controller with better performance characteristics and possibly in context extraction where the network state table look-up can block for a long time.

We think there are two plausible explanations to the poor worst case performance for reinjection. Either the Windows Filtering Platform injection handle queue is full and applying back-pressure against our injection requests, or we are being negatively impacted by process scheduling. The injection handle queue filling up seems unlikely as the queue should be getting drained at about line rate for the system, and it is very unlikely our userspace application is generating traffic that quickly. Instead, we believe that our OpenFlow Agent is getting preempted in the middle of servicing an I/O request, causing artificially bad results. The Windows process scheduler will context-switch away from the application when it blocks on a call to `WinDivertSend`.

WinDivert will then receive a request and service it, calling into the Windows Filtering Platform and waiting for an asynchronous callback indicating completion of the injection. WinDivert will then complete the applications request, unblocking it. At this point the scheduler *can* resume execution of the OpenFlow agent, but it does not *have* to. Another high priority process could be running, and several other high priority I/O requests and interrupts may need to be serviced first. In this way we could see in really pathological cases a multi-second delay while blocking on `WinDivertSend`. We discuss possibilities for mitigating this in subsection 6.1

6 Concluding Remarks

Distributed firewalls allow network controllers to block malicious traffic and prevent attacks. However, the information given to these controllers is limited, as the context for the traffic is not normally specified. WinSight provides a tool which enables an OpenFlow controller to make a more informed decision when filtering network flows by including host context with every PacketIn. We were able to make a basic OpenFlow agent, collect contextual information, and provide that to a custom PacketIn handler in the POX OpenFlow controller. Excluding statistical outliers, our implementation incurs reasonable overheads and applies fine-grain filter rules informed by contextual information. Coarse grained contextual ACLs can be enforced in a similar fashion as fine-grain OpenFlow rules.

6.1 Future Work

There are clear directions for future work on WinSight on Windows. The current implementation suffers from poor performance in the worst case, but there are several possible strategies for mitigating this. The adaptation of the Linux subsystem for Windows 10 may provide the ability to use software we were unable to port. Policy with host context data also needs exploration: what kinds of new applications and monitoring does it allow? Finally WinSight could make several existing OpenFlow-SDN applications more practically applicable by moving filtering burden onto end hosts. All of these avenues can and should be explored.

Performance of the existing WinSight prototype on Windows is not quite ready for a trial deployment. We have shown that in the median case performance is actually quite good, but in the worst case performance is bad, with the two separated by a significant four orders of magnitude (when blocking). We believe that this is caused by pathological user-space process scheduling behavior and the use of a synchronous blocking programming model in our prototype. The Windows process scheduler does not treat our user-space filtering prototype with a high enough priority and causes us to see multi-second delays in packet reinjection. The blocking programming model means that we ignore new packets arriving on the divert socket and traffic gets unnecessarily blocked. With the switch to the asynchronous model, the performance increased significantly, but we cannot confidently say that the performance issue seen in the synchronous model was not merely pushed into the background.

We think that adopting an asynchronous event-based model will reduce the amount of time that the OpenFlow agent spends being blocked, and make this inherently concurrent application easier to reason about. All of the interactions with WinDivert can be done asynchronously with `WinDivertRecvEx` and `WinDivertSendEx`, allowing us to relinquish control to callbacks from the kernel. The callback for `WinDivertRecvEx` places a packet on the queue for flow table filtering and possible escalation. A thread or thread poll reads from the filtering queue and either escalates the packet or puts it in a queue for reinjection. The callback for `WinDivertSendEx` removes an enqueued packet from the reinjection queue. This architecture would avoid blocking except on the internal queues, which will be notified by external events from the kernel. This mitigates the poor performance seen when `WinDivertSend` calls block for a long time, but appears to not fully resolve the issue seen by the blocking `WinDivertSend` function.

Performance can further be improved by filtering in the kernel and avoiding the user-space scheduler and copying costs all together. This can be achieved with simple improvements to WinDivert. The primary limitation of WinDivert is that it has a static filter described by a filter. While this fits the exploratory divert socket use case quite well, it does not fit for a dynamic OpenFlow flow table and filter.

We can extend WinDivert to be dynamic without modifying the kernel driver. Recall that in the Windows Filtering Platform, all sublayers in a filtering layer are evaluated, but a sublayer is traversed in priority ordering until a filter matches and returns a decision. We can perform dynamic filtering simply by placing WinDivert in the lowest priority in a sublayer and inserting filters in the same sublayer. This filter management can be done entirely from user-space, meaning that this approach does not even require significant kernel-level development.

However, this approach will not allow for more complicated forms of OpenFlow actions, only dropping traffic. For the initial purposes of WinSight, this should be sufficient, and future work could also develop a feature-rich OpenFlow table filtering callout for the Windows Filtering Platform. Such a system should mimic the architecture of Open Virtual Switch [27] and start with an exact match “microflow” cache. This could similarly be placed in higher priority than the divert socket callout. If this solution has poor performance because of cache thrashing, the OVS “megaflow” cache could similarly be inserted between the “microflow” cache and the divert socket.

While we demonstrated deep packet inspection capabilities with our system, at present they are rather limited. The traditional OpenFlow model only generates a PacketIn for the first message in a flow, which keeps network usage and computational costs down at the expense of visibility into traffic. Our current system would only be able to examine the first packet of a flow, and since most traffic in a network is carried over TCP and the first packet in a TCP flow is the empty SYN message, there is no content from the stream available for analysis. If deep packet inspection is required, a practical approach could be to combine the host-based enforcement and context of WinSight with the distributed enforcement and centralized policy decision model of EnforSDN. In EnforSDN, a traditional network appliance makes policy decisions for the network, but the network enforces the policy [7].

Further, the context gathering or transmission portion of WinSight currently is not optimized. For example, a virus could name itself `winword.exe` and the current dataset would have nothing to validate or discriminate between Microsoft Word and a file masquerading as Microsoft Word. Including a signature for executables could distinguish a malicious file from a Microsoft- or otherwise-certified binary. Additionally, the protocol for storing and transmitting the context can greatly increase the amount of data required to transmit a PacketIn, due to the ability to store up to 255 characters in fields such as paths and executable names. A compression scheme could likely reduce the size of data that is required to be sent.

6.2 Trial Deployment

For a trial deployment of WinSight, the POX OpenFlow controller would be set up on a LAN, with the WinSight agent installed on computers connected to the LAN. WinSight creates the ability to examine packet context information, but it does not come with any rules on context by default. Rules can be implemented on the controller by adding to the `_handle_PacketIn` function in the POX module. For instance, in order to add a rule which blocks packets from a program with the window title "badprogram", an if-statement can be placed which checks the title portion of the context data for the string "badprogram". If the string exists in the title, a return statement can be added to prevent the controller from sending back a PacketOut message. Without this PacketOut message, the agent will never re-inject the packet into the network stack, successfully blocking the traffic.

WinSight does not require every computer to have an agent, but those without the agent will not benefit from the security enhancements. As such, it is possible for a computer to operate on the network normally without the agent. If this computer sends traffic to another computer with the WinSight agent installed, the traffic will be evaluated by the controller. With this in mind, it is possible to do an incremental deployment on a LAN, where only some computers have WinSight installed. This is desirable because WinSight could be deployed on a subset of the available systems without requiring full commitment. The ability to be partially deployed is also useful for when computers connecting to the LAN are not permanent residents, such as laptops and smart-phones. With this functionality, WinSight can protect core computers with sensitive data, while leaving the network accessible to temporary connections.

To test the trial deployment, there are two high-level attack vectors to consider: from within the network, and from beyond the network. An attack within the network involves the source of the attack to be connected to the LAN with WinSight. The attacker would attempt to spread malware through the LAN while WinSight would have to prevent it. This test should be done both with and without WinSight installed on the attacking computer. In a real-world scenario, this would likely be carried out through corporate espionage or social engineering, such as malware left on a "lost" flash drive. An attack from beyond the network is similar to the aforementioned test, except the attacking computer attempts to reach the LAN from across the Internet. This attack cannot be carried out with WinSight installed on the attacking computer, as the attacker is not connected to the LAN. The attacker would likely attempt similar attacks against WinSight and legacy networks.

A WinDivert Installation

I. Enable test driver signing: `bcdedit.exe -set testsigning on`
This allows a locally generated certificate to be used without being signed by some trusted root certificate authority.

II. Obtain WinDivert from https://www.reqrypt.org/windivert.html

III. Create a certificate and store for the WinDivert driver: `MakeCert -r -pe -ss WinDivertStore -n "CN=WinDivert" WinDivert.cer`.

IV. Add the new certificate to the machine's trusted root store: `CertMgr /add WinDivert.cer /s /r localMachine root`.

V. Sign WinDivert driver with certificate used in last command: `SignTool sign /v /s WinDivertStore /n WinDivert WinDivert[32|64].sys`

VI. Restart machine. On restart driver will be loaded.

B Alternate Approaches

This appendix discusses optimizations and alternative approaches that we believe are noteworthy.

One of our major concerns regarding our design and implementation is that every packet must cross the boundary between kernel and user space. While the performance timings reveal this is not as severe as expected, it is still significant. Because of the nature of kernel memory, the kernel/user-space crossover we rely upon means that a packet must be fully copied into user space, and then copied back to kernel space again. Including the step to get the packet from a network device into kernel space and back out, there are four total copies made, and our approach has doubled the number. Copying packets in memory can be expensive, so we expect to see this affect our throughput and latency, as discussed in Figure 5.

We do believe that the fundamental design, using OpenFlow with context in-band on host agents, allows for more flexible control of network policy. The following approach merely changes the way our host agent is implemented.

B.1 Kernel Fast Path

To overcome the throughput and latency limitations created by having to buffer packets between the user-space OpenFlow Agent and kernel-space WinDivert driver, we suggest that the agent be split in half, providing a "slow path" and a "fast path."

The "slow path" sits in userspace and behaves similarly to our current OpenFlow Agent. If it does not have an entry for a flow, it generates a PacketIn and the controller responds with a FlowMod or a PacketOut as normal. When the agent receives the FlowMod, in addition to updating the user space flow tables, it additionally communicates the flow information to the kernel space "fast path."

The kernel space "fast path" avoids the expensive copying of packets by keeping a cache of flow information and only communicates with the user space agent on a cache miss. This means that there will still be at least user to kernel space latency on the startup of a flow, but once it is permitted or blocked it never has to leave the kernel space fast path. This echoes the construction of most OpenFlow switches, which have a software slow path for handling OpenFlow table misses and a fast path consisting of specialized hardware and TCAM. It also echoes the design of Open Virtual Switch, a host based OpenFlow agent for the Linux operating system whose design is described in [27].

We believe that though modification of the WinDivert driver could feasibly provide this method, it would not be significantly easier than designing a module by hand. However, as the next section details, we had many challenges in our attempts to do so.

B.2 Kernel Driver

We attempted to develop our own kernel driver using the community edition of Visual Studio 2015 in place of WinDivert. However, we ran into significant challenges setting up the debugging environment, whether using physical media, network transmission, or other methods, both on virtual and physical machines. This method will likely perform better than the WinDivert user-space interaction, but due to time constraints and the challenges we ran into, we decided to abandon this approach. One of the challenges we ran into while delving into the Windows kernel and networking API was finding that the MSDN documentation is inconsistent and often impossible to replicate listed feature sets. We also contacted a student who had successfully completed a Windows driver as part of his project the about two months prior to our efforts. He was unable to successfully recreate his kernel development environment, at which point we decided to proceed with other options.

The library we primarily attempted to develop in was the Windows Filtering Platform (WFP). Additionally, we researched Network Driver Interface Specification (NDIS), but ultimately found it to be too low-level to be useful. What follows is what we consider the relevant knowledge in the WFP API obtained through our research for any other projects looking for guidance in this domain.

The Windows Filtering Platform (WFP) is a set of APIs and services that allow the creation of network filtering applications, which can be used for the OpenFlow Agent datapath. WFP operates on multiple layers of the network stack and allows the interception of packets before they are transmitted to the network or pushed up to the receiving application. The WFP API allows control via userspace applications, but will do the majority of its processing in kernel space. This means packets do not have to be passed into or out of kernel space for processing, making it inherently faster than the system we chose to use.

A session to WFP's filter engine can be opened by calling the function `FwpmEngine-Open0`, which returns an engine handle. A filter can be added by calling `FwpmFilterAdd0`, which takes in the engine handle and a filter defined by the struct `FWPM_FILTER0`. The arguments to `FWPM_FILTER0` contains a layer key, an array of filter conditions, and an action, amongst other fields. The layer-key is the GUID of the layer where the filter resides, such as `FWPM_LAYER_INBOUND_IPPACKET_V4`. The filter conditions are defined by the struct `FWPM_FILTER_CONDITION_0`, which contains the field to be tested, a match type, and the value for the field to be matched against. The action is defined by the struct `FWPM_ACTION0`, and can be set to block or permit packet flows. The action can also be set to invoke a callout on the packet, which can be either a built-in or user-written function.

To implement WinSight's datapath using WFP, callouts for sending packet flows to and receiving packet flows from the controller are needed. Doing so will avoid sending decision requests on traffic destined for the controller, to the controller, as this will create a cycle. Until a response is received from the controller, the function `FwpsPendOperation0` can be used to postpone a classification decision on a packet flow. Once a packet flow is ready to be fully processed, packet operations can be resumed by calling `FwpsCompleteOperation0`. If the packet flow is deemed safe by the controller, a filter with a permit action will be installed on that flow. If the packet flow is deemed dangerous, a filter with a block action can be installed. When a flow is deemed dangerous, packets from that flow can be silently dropped to prevent the origin machine from detecting a block.

B.3 xDPd as an OpenFlow Agent

As a final mention, we looked into using an existing OpenFlow agent instead of writing our own from scratch. The eXtensible DataPath Daemon (xDPd), a project of the Berlin Institute for Software Defined Networks (BISDN), aims to provide a common set of tools for creating OpenFlow switches and is intended to ease the implementation burden when trying to prototype a new OpenFlow device or perform research [8]. In that regard it is a perfect match for our needs. It consists of three major parts from the Revised OpenFlow Library (rofl): rofl-common which handles an OpenFlow channel and turns messages into internal events, rofl-pipeline which implements a software OpenFlow switch with flow tables, and rofl-hal which provides API hooks for a hardware abstraction layer. With xDPd, we need only write platform support for Windows that hooks into the rofl-hal API, and we would get a full OpenFlow switch. Unfortunately xDPd makes numerous assumptions about the target system, namely that it is a POSIX system, and worse has certain Linux APIs exposed. These assumptions made it difficult to bring xDPd into the Windows environment; significant effort would have to be invested to port it for actual use. However, rofl-pipeline is largely platform independent and implements fast flow table search algorithms. If developing a fast

OpenFlow switch implementation, it may be a good starting point. However, we did not believe it was worth the time for this version, as we were looking to prove the idea is feasible and the efficiency difference did not warrant the expenditure of effort required. With the recent addition of the Linux subsystem to Windows 10 insider builds, this may be a worthwhile topic to investigate.

References

[1] *2015 Cost of Cyber Crime Study: Global.* 2015. URL: http://www.statista.com/statistics/293274/average-cyber-crime-costs-to-companies-in-selected-countries/.

[2] *2015 Cost of Cyber Crime Study: United States.* 2015. URL: http://www.statista.com/statistics/293256/cyber-crime-attacks-experienced-by-us-companies/.

[3] *2015 ITRC Breach Stats Report.* 2015. URL: http://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed/.

[4] William A Arbaugh, David J Farber, and Jonathan M Smith. "A secure and reliable bootstrap architecture". In: *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on.* IEEE. 1997, pp. 65–71.

[5] basil. URL: https://www.reqrypt.org/windivert.html.

[6] Steven M Bellovin. "Distributed firewalls". In: *Journal of Login* 24.5 (1999), pp. 37–39.

[7] Yaniv Ben-Itzhak et al. "EnforSDN: Network policies enforcement with SDN". In: *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on.* IEEE. 2015, pp. 80–88.

[8] BISDN. 2014. URL: http://xdpd.org.

[9] Rodrigo Braga, Edjard Mota, and Alexandre Passito. "Lightweight DDoS flooding attack detection using NOX/OpenFlow". In: *Local Computer Networks (LCN), 2010 IEEE 35th Conference on.* IEEE. 2010, pp. 408–415.

[10] Martin Casado et al. "Ethane: Taking Control of the Enterprise". In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications.* SIGCOMM '07. Kyoto, Japan: ACM, 2007, pp. 1–12. ISBN: 978-1-59593-713-1. DOI: 10.1145/

1282380.1282382. URL: `http://doi.acm.org/10.1145/1282380.1282382`.

[11] Cisco. URL: `http://www.cisco.com/c/en/us/about/security-center/virus-differences.html`.

[12] Jake Collings and Jun Liu. “An openFlow-based prototype of SDN-oriented stateful hardware firewalls”. In: *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*. IEEE. 2014, pp. 525–528.

[13] Mehiar Dabbagh et al. “Software-defined networking security: pros and cons”. In: *Communications Magazine, IEEE* 53.6 (2015), pp. 73–79.

[14] Hongxin Hu et al. “FLOWGUARD: building robust firewalls for software-defined networks”. In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, pp. 97–102.

[15] Zhiyuan Hu et al. “A comprehensive security architecture for SDN”. In: *Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on*. IEEE. 2015, pp. 30–37.

[16] Kenneth Ingham and Stephanie Forrest. “A history and survey of network firewalls”. In: *University of New Mexico, Tech. Rep* (2002).

[17] Sotiris Ioannidis et al. “Implementing a distributed firewall”. In: *Proceedings of the 7th ACM conference on Computer and communications security*. ACM. 2000, pp. 190–199.

[18] Debojyoti Dutta Kuchan Lan Alefiya Hussain. “Effect of Malicious Traffic on the Network”. In: USC, 2003, pp. 1–5.

[19] Butler Lampson et al. “Authentication in distributed systems: Theory and practice”. In: *ACM Transactions on Computer Systems (TOCS)* 10.4 (1992), pp. 265–310.

[20] M. Lelarge. “Economics of malware: Epidemic risks model, network externalities and incentives”. In: *Communication, Control, and Computing, 2009. Allerton 2009. 47th Annual Allerton Conference on*. Sept. 2009, pp. 1353–1360. DOI: `10.1109/ALLERTON.2009.5394516`.

[21] Douglas C. MacFarland and Craig A. Shue. “Internal working document”. A work at WPI exploring context extraction on Linux by hooking into GTK+ and its applications for system call trace log analysis.

[22] Nick McKeown et al. “OpenFlow: Enabling Innovation in Campus Networks”. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: `10.1145/1355734.1355746`. URL: `http://doi.acm.org/10.1145/1355734.1355746`.

[23] Syed Akbar Mehdi, Junaid Khalid, and Syed Ali Khayam. “Revisiting traffic anomaly detection using software defined networking”. In: *Recent Advances in Intrusion Detection*. Springer. 2011, pp. 161–180.

[24] Microsoft. 2016. URL: `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366510(v=vs.85).aspx`.

[25] MinGW. 2016. URL: `mingw.org`.

[26] Hilarie Orman. “The Morris worm: A fifteen-year perspective”. In: *IEEE Security & Privacy* 5 (2003), pp. 35–43.

[27] Ben Pfaff et al. “The design and implementation of open vswitch”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 117–130.

[28] S. Scott-Hayward, G. O’Callaghan, and S. Sezer. “Sdn Security: A Survey”. In: *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. Nov. 2013, pp. 1–7. DOI: `10.1109/SDN4FNS.2013.6702553`.

[29] Sandra Scott-Hayward, Gemma O’Callaghan, and Sakir Sezer. “SDN security: A survey”. In: *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*. IEEE. 2013, pp. 1–7.

[30] Jeffrey Shirley and David Evans. “The user is not the enemy: Fighting malware by tracking user intentions”. In: *Proceedings of the 2008 workshop on New security paradigms*. ACM. 2009, pp. 33–45.

[31] Curtis R. Taylor et al. “Contextual, Flow-Based Access Control with Scalable Host-based SDN Techniques”. In: *INFOCOM*. IEEE. Apr. 2016.

[32] Huijun Xiong et al. “User-assisted host-based detection of outbound malware traffic”. In: *Information and Communications Security*. Springer, 2009, pp. 293–307.