Enabling Practical Software-defined Networking Security Applications with OFX

John Sonchack
University of Pennsylvania
jsonch@seas.upenn.edu

Adam J. Aviv
United States Naval Academy
aviv@usna.edu

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

Jonathan M. Smith
University of Pennsylvania
jms@cis.upenn.edu

***Abstract*—Software Defined Networks (SDNs) are an appealing platform for network security applications. However, existing approaches to building security applications on SDNs are not practical because of performance and deployment challenges. Network security applications often need to analyze and process traffic in more advanced ways than SDN data plane implementations, such as OpenFlow, allow. Much of an application ends up running on the centralized controller, which forms an inherent bottleneck. Researchers have proposed application specific modifications to the underlying data plane to gain performance, but this results in a solution that is not deployable as it requires new switches and does not support all network security applications. In this paper, we introduce OFX (the OpenFlow Extension Framework) which harnesses the processing power of network switches to enable practical SDN security applications within an existing OpenFlow infrastructure. OFX allows applications to dynamically load software modules directly onto unmodified network switches where application-dependent processing/monitoring can execute closer to the data plane at a rate much closer to line speed. We implemented OFX modules for security applications including Silverline (ACSAC'13), BotMiner (Sec'08), and several others motivated by the custom OpenFlow extensions in Avant-Guard (CCS'13). We evaluated OFX on a Pica 8 3290 switch and found that processing traffic in an OFX module running on the switch had orders of magnitude less overhead than processing traffic at the controller. OFX increased the performance of the evaluated security application by 20-40x as compared to standard OpenFlow implementations and up to 1.25x when compared to middlebox implementations running on dedicated servers. This is all achieved without the need for additional or modified hardware.**

I. Introduction

Software Defined Networking (SDN) has seen widespread adoption by many companies such as Google, Facebook, and Microsoft. Naturally, the security research community has recognized the potential to leverage programmable networks for novel security applications [24], [30], [15]. Unfortunately, existing approaches to building network security applications on top of SDNs are not *practical* – they either limit performance or deployability.

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23309

A. Limitations of SDN-based Security Applications

Performance Limitations Network security applications often require processing and analysis techniques that are more advanced than SDN data planes allow. OpenFlow, the de-facto SDN standard for controlling switches, has many noted data plane limitations [16], [35], for example. Due to these limitations, SDN based security applications must implement much of their functionality in the *control plane* (*i.e.*, at the centralized network control server that manages the data plane switches). Fresco [34] takes this approach, and provides a framework that simplifies the development of control plane network security applications. However, implementing functionality in the control plane hinders performance because the communication channel between the data plane and control plane is a bottleneck that adds latency and limits the amount of traffic that the security application can process. It also limits scalability because there are usually far fewer controllers than switches in a network.

> *For SDN to be practical for security applications, the data plane needs to support more advanced functionality.*

Deployment Hurdles Outside of the security domain, vendors have added features to switches with *software* based data plane extensions (commonly referred to as slow path processing [29]), to provide additional functionality close to the hardware-based forwarding path. For example, Cisco switches run the embedded event manager (EEM) [3] which can trigger minor reconfigurations based on a preset selection of events, and BigSwitch Network pushed the ARP handling from its Floodlight [8] SDN control into its Switch Light operating system that runs on whitebox SDN switches [20]. Avant-Guard [35] demonstrated that a similar approach can also be effective for security applications by designing custom switch data planes that provide new functionality. Unfortunately, these approaches have two drawbacks that make them impractical. First, they define extensions that are point solutions with only certain applications in mind. Security applications are diverse in functionality, and unless an extension is programmable, it is not likely to benefit many applications besides the ones in mind when it was designed. Furthermore, for network operators to use a custom data plane, or vendor specific extension, they must deploy new hardware and/or software into the network which may be incompatible with the existing SDN infrastructure (hardware and software).

For SDN to be practical for security applications, the added data plane functionality should be highly programmable and work within existing SDN technology for both hardware (switches) and software (controllers).

B. Our Proposal: OpenFlow Extension Framework

In this paper we introduce an *OpenFlow Extension Framework* (OFX), which makes SDN practical for network security applications by allowing them to extend most OpenFlow enabled switches with custom functionality. To accomplish this task, OFX leverages switches' pre- existing general purpose hardware and software. Most modern switches run a small operating system [10], [5] (typically Linux- based) with computational capabilities that are currently being under utilized. OFX opens these resources up to network security applications by allowing them to install OFX software modules that perform application specific processing/monitoring tasks directly on the switch, closer to the data plane. This approach makes many security applications much more practical using existing hardware because it reduces the overhead of performing complex packet processing and the amount of interaction between the data and control plane.

A typical OFX enabled application, *e.g.*, a DDoS detector, defines custom switch functionality , *e.g.*, threshold tests for the rate of traffic flow *à la* Avant-Guard, that can be deployed onto switches within the network. Once executing in the switch, the OFX module can integrate with the existing OpenFlow framework to monitor and process traffic, install forwarding rules, and communicate with the centralized controller (or other servers on the network) if needed. Functionally, OFX provides new control features and programmability between the data plane and the control plane on the switch where packets can be processed without having to traverse the long path to the controller.

We implemented and evaluated OFX-based applications in a testbed with a widely deployed model of a hardware OpenFlow switch (Pica 8 3290) to measure the overhead and benefits of using OFX. We found that the overhead to process packets in an OFX module running on the switch was orders of magnitude lower than the overhead to process packets at the OpenFlow controller. Furthermore, OFX modules provided similar benefits and functionality as previously proposed switch extensions, such as Avant-Guard, but with greater deployability because OFX modules run on unmodified hardware switches. Finally, by deploying custom OFX modules onto the switch, the capacity of the network security applications we tested increased by $20x - 40x$ when compared to using an OpenFlow control plane implementation alone. OFX had up to $1.25x$ higher capacity when compared against middlebox implementations running on dedicated servers. These gains are possible with adding any hardware to a SDN network or modifying its low level software.

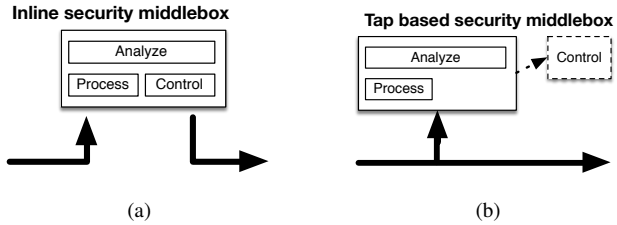


Fig. 1: Network security using a middlebox.

C. Contributions

To summarize, this work's contributions are as follows:

- Introduction of a novel SDN framework that allows data plane extensibility within existing SDN infrastructures.
- Demonstration of OFX capability by implementing three security applications using OFX for DDoS protection, network based taint tracking declassification, and botnet detection.
- Evaluation of OFX on real OpenFlow switching hardware demonstrating that it meets performance and deployment requirements.

We also wish to stress that this is the first proposal, to the best of our knowledge, that allows security applications to extend existing, unmodified OpenFlow switches in a functional and high-performance manner. Further, OFX is a general platform and can be used for a variety of applications beyond those described herein, and may provide benefits to the wider SDN community.

II. Background and Motivation

OFX builds on many years of research in network security. Here we describe several network security applications and how they would be traditionally implemented. We then discuss how these applications could benefit from SDN in theory and the drawbacks to using SDN in practice. From this, the motivation for practical SDN based security applications with OFX will follow.

A. Network Security Applications

Many network-based security applications can be decomposed into three parts, as illustrated in Figure 1: *processing*, *analysis*, and *control*. The processing component of the security application performs measurements and extracts features from network traffic. Next, the application must analyze the data, come to a decision point, and draw conclusions, e.g., is there nefarious activity or not? Finally, the control portion of the logic handles traffic with respect to the analysis results.

Traditionally, security applications are deployed onto middleboxes – specialized equipment or software that exists in-line with network activity or taps into the network to receive copies (or samples) of network traffic. Both setups can perform processing and analysis, but while in-line middleboxes can affect

network traffic directly (e.g., drop traffic like a firewall), tap-based applications can only create alerts for other equipment or network operators to take action on (e.g. an intrusion detection system generating alerts for a network operator).

As a running example of the kinds of security applications we wish to support with OFX, we outline middlebox implementations of three sample security applications:

- *DDoS Detection:* Consider a DDoS detector that effectively functions as a monitor of bandwidth usage at a host (e.g., web server) and will alert a network operator when thresholds are exceeded. If implemented as a tap-based application, it would: 1) measures the length of each packet sent to the monitored host (*processing*); 2) calculates the bandwidth usage and determines if it is above the threshold (*analysis*); 3) alert the network operator if the bandwidth is above the threshold (*control*).
- *Network Taint-Tracking Declassifier:* As an example of an inline network security application, we consider Silverline [32], a taint tracking system that contains a network based declassifier which ensures a web client has permission to receive the data that it has requested from a locally monitored web server before the packets containing data leave the local network. Implemented as an inline application, the declassifier would: 1) extract the taint tag from each packet (*processing*); 2) check the tag against the client's permissions stored in a Silverline database (*analysis*); 3) if the client has the permission to view the data in the packet, allow it to pass, otherwise drop the packet (*control*).
- *Botnet Detection:* As a third example, consider intrusion and botnet detection as proposed by Botminer [22], a network monitoring application that detects bots in a network by clustering flow records. Again, implemented as a middlebox tap application, it would: 1) extract flow records from network traffic (*processing*); 2) cluster the flow records to detect hosts with similar, suspicious communication patterns and decide if any hosts are bots (*analysis*); 3) alert the network operators of detected bots, so that they can take appropriate measures (*control*).

B. Using SDN to Enhance Security Applications

SDNs provide a standardized interface for an application to remotely manage switch forwarding tables. A security application implemented as an SDN application, which Figure 2a illustrates, can push its *traffic control logic* into the data plane by installing forwarding rules. However, more advanced processing and analysis logic cannot generally be encoded using forwarding rules, and must run at the controller. For example, if the declassifier was implemented as a SDN control application, it could check the permissions on the first packet of a flow and then, if the packet's permissions are correct, install a rule on the switch to forward the flow's remaining packets without sending them to the control application. If the DDoS or botnet detectors were implemented as SDN control applications, they could install a forwarding rule for each traffic flow, poll the switches for statistics about the traffic flows, and then modify the forwarding rules to block or redirect malicious flows.

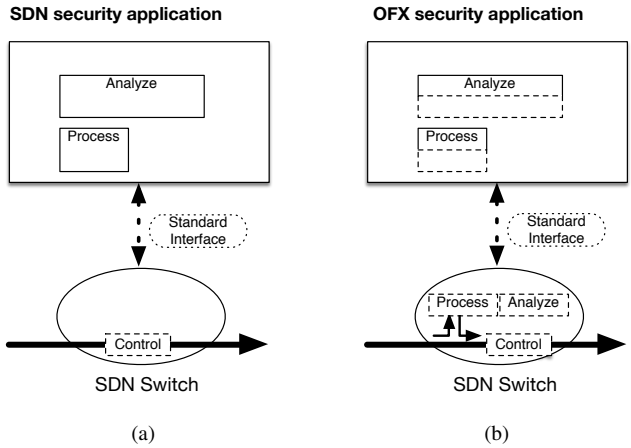


Fig. 2: Network security using an SDN, and OFX.

However, there are two important drawbacks to implementing security applications at the controller. First, there is an issue of scalability as there are far fewer controllers in a network than switches (often times only one controller). As the network grows, the controller is tasked with more and more work. More fundamentally, there is a larger challenge of performance regarding the controller. The path between the data plane and control plane has many bottlenecks that can lead to high latency and greatly restrict network bandwidth and throughput. For example, our measurements in Section VI-B show that for a Pica 8 3290, the control path can add upwards of 1 second of delay to packets and can handle fewer than 1000 packets per second.

To overcome the scalability and performance challenges, Avant- Guard [35] proposed to embed new packet processing functions into the forwarding logic of software switches, such as Open vSwitch [11] or Click [27]. While extending the data plane adds scalability and performance, there are several large drawbacks to existing data plane extensions. First, they cannot be deployed onto hardware OpenFlow switches with data planes implemented as ASICs (Application Specific Integrated Circuits). Second, existing extensions are only designed for specific applications (e.g. the Avant-Guard extensions may not improve the performance of the example declassifier or botnet detector applications).

III. Introducing OFX

Our OpenFlow Extension Framework (OFX) is a practical SDN platform that provides both high performance and deployability. It leverages the processing power of the switches themselves to run security application logic and mechanisms that would otherwise run on the controller, and allows network security applications to define their own custom data plane functionality that works within the existing SDN hardware and software infrastructure. In this section, we provide a high-level description of OFX and how the example security applications can leverage OFX. We expand on the OFX architecture in

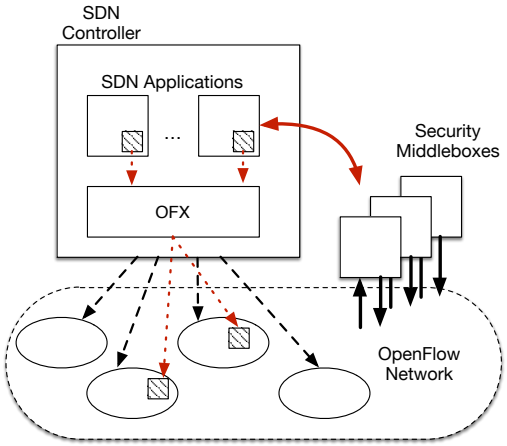


Fig. 3: The OFX architecture from a network-wide perspective.

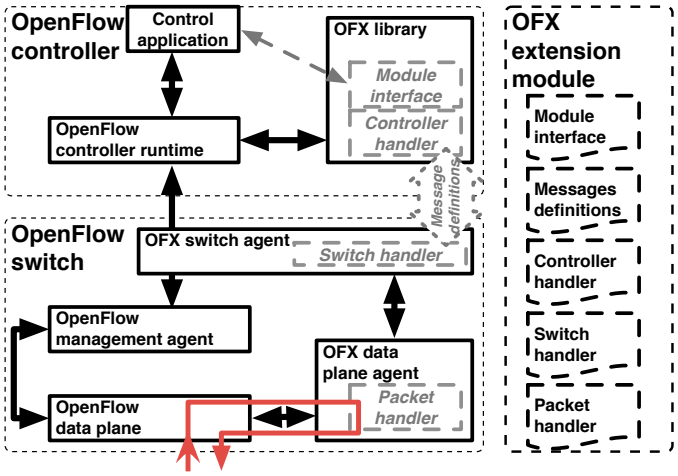


Fig. 4: Adding security functionality with OFX, which loads extension modules onto agents that run on a switch's general purpose hardware. The of packets processed by an extension module is shown in red.

Section IV, and elaborate on the design details of the example security applications in Section V.

A. OFX Design

As illustrated in Figure 2b, OFX allows a network security application to not only move its traffic control logic to the data plane (as with SDN solutions), but also parts of the processing and analysis logic. Although OFX extends switches, it remains deployable because it runs on the general purpose processors that all OpenFlow switches have. Router vendors have used these resources to add pre-defined functionality to their devices [20], [29], [3]. OFX provides a framework that opens up the resources to enable custom, application-defined data plane functionality.

At a network wide level, which Figure 3 illustrates, OFX allows SDN controller applications to install software modules into the data plane. Further, these SDN applications can interact with the rest of the security infrastructure, enabling traditional security middleboxes to make requests to install functionality into the data plane that enhances their performance. This builds on the design proposed by PANE [19], in which end hosts could request bandwidth reservations from the network.

B. OFX Application Integration

All of the example applications described in Section II could be redesigned as OFX applications that push custom functionality into the data plane to improve performance.

DDoS Detection The DDoS detection control application could be refactored so that instead of polling the switch for flow statistics, it installs a module onto the switch that monitors traffic and only sends an alert to the controller if it detects the DDoS condition. This refactoring requires a minimal amount of changes to the SDN application itself, as the details can be implemented in an OFX module that provides a simple abstraction to the controller. We demonstrate that using an OFX trigger alert module allows for much quicker DDoS detection times in Section VI.

Network Taint-Tracking Declassifier The network taint-tracking declassifier could connect with an OpenFlow control application that uses OFX, and request that the application load an OFX module with declassification logic onto the switch. The declassifier could then load permissions onto the switch via a channel maintained by the OpenFlow control application. The declassifier module would check permissions locally on the switch, without having to send packets from each flow to the centralized controller or a declassification middlebox.

Botnet Detection For botnet detection, instead of tapping a network switch to receive a copy of each packet and then doing the preprocessing and analysis on a middlebox server, a botnet detector middlebox could connect with an OFX aware OpenFlow control application and request that a pre-processing function be loaded onto the switch. The switch could then send batch updates containing the processed feature vectors to the middlebox via the controller instead of sending each packet to the middlebox for processing. In Section VI, we demonstrate that the OFX approach is effective, especially for high bandwidth scenarios.

IV. OFX Architecture

Figure 4 depicts the OFX framework. *OFX modules* specify new security functionality to add to the network, and an interface for control applications to access it. The *OFX library* contains functions that OpenFlow control applications can use to load and manage OFX modules. An *OFX switch agent* manages the modules running on the switch and handles communication between the controller and switch components of modules. Finally, the *OFX data plane agent* executes a module's packet handlers (*i.e.* packet processing functions). In the remainder of this section, we describe these components of OFX in greater detail.

Function	Description
`MessageSwitch(Message, SwitchID)`	Sends a message from a module's controller component to its switch component.
`MessageController(MessageContent)`	Sends a message from a module's switch component to its controller component.
`AddInterceptFlow(MatchPattern)`	Register a module's packet processor to receive packets that match the given OpenFlow pattern.
`AddTapFlow(MatchPattern)`	Register a module's packet processor to receive copies of packets that match the given OpenFlow pattern.
`AddNegativeInterceptFlow(MatchPattern)`	Register a module's packet processor to **not** intercept packets that match the given OpenFlow pattern.
`AddNegativeTapFlow(MatchPattern)`	Register a module's packet processor to **not** tap packets that match the given OpenFlow pattern.
`GetFlowStats(FlowIdList)`	gets statistics for a list of flows that the module has added through OFX.
`RemoveFlows(FlowIdList)`	Instructs OFX to remove an intercept, tap, or negation flow.
`SendPacketIn(Packet)`	Sends a packet to the control plane, encapsulated in a standard OpenFlow `packet_in` message.

TABLE I: The OFX Module API.

Function	Description
`LoadModule(FileName)`	loads an OFX module and returns a pointer to its control interface.
`PushModule(ModuleID, SwitchID)`	installs a module onto a switch.

TABLE II: The OFX Controller API.

A. OFX Modules

An OFX module defines new security functionality for OpenFlow switches and an interface that OpenFlow control applications can use to invoke the functionality. Modules can be dynamically loaded and installed onto switches by control applications. To write an OFX module, a developer needs to define:

- **A module interface**, which defines new functions that an OpenFlow control application can call once the module is loaded onto the controller and switches in a network. Typically, module interface functions will simply generate and send module messages from the controller to one or more switches that have the module loaded.
- **Control messages** both for controller to switch signaling (*i.e.* to invoke new switch functionality), as well as switch to controller signaling (*i.e.* to provide replies to the controller). An OFX module can send messages between its controller and switch components using API functions in the OFX library or OFX agent.
- **A switch handler** that will be called whenever a message defined in the module reaches a switch that has the module loaded.
- **A controller handler** that will be called whenever a message defined in the module reaches the controller.
- **A packet handler** that contains any code that the module needs to run on individual packets. When a packet handler is first loaded onto a switch, it will not receive any packets. The module must register to receive packets from specific flows, using OFX API functions that the OFX switch agent provides.
- **Startup functions** that contains any code the module needs to run when it is loaded onto controllers and switches.

The OFX Module API

The OFX Module API, summarized in Table I provides functions that an OFX module can use to send messages between its controller and switch components, select which packets its handlers receive, and interact with the OpenFlow data and control plane in limited ways.

The `MessageSwitch` and `MessageController` functions allow an OFX module to send messages from its controller component (*i.e.* the module code running in the controller) to its switch component (*i.e.* the module code running in the OFX agents on switches), and vice versa.

The `AddInterceptFlow` and `AddTapFlow` functions allow an OFX module to register its packet handler to receive packets as they pass through the switch's data plane. Both of these functions accept an OpenFlow match pattern as input, which can specify one or more Ethernet, IP, UDP or TCP header fields, and permits wildcard entries.

The `AddInterceptFlow` function causes OFX to install an OpenFlow rule that redirects packet to a module's packet handler *before* they reach the main forwarding table of the switch. The `AddTapFlow` function causes OFX to install an OpenFlow rule that sends a *copy* of all matching packets to the module's packet processor.

The `AddNegativeInterceptFlow` and `AddNegativeTapFlow` functions allows an OFX module to specify that it does *not* want to receive packets belonging to a particular flow. This functionality is useful when an OFX module only needs to process the first few packets of each flow and does not want to waste resources processing subsequent packets. For example, our implementation of the Silverline [32] declassifier, which we describe in Section V, needs to check the first packet of every flow leaving a monitored web server. The declassifier can implement this functionality to monitor a particular server, say 10.1.1.1, by first calling `AddInterceptFlow(src=10.1.1.1)`. Then, after it checks the permissions on the first packet of each new flow from 10.1.1.1, it can call `AddNegativeInterceptFlow` with the flow key as the match pattern, so that the flow's remaining packets proceed directly to the main forwarding table of the switch instead of going through OFX.

The `GetFlowStats` function allows an OFX security module to get packet and byte count statistics for any of the flows that it registered using any of the above method. The OFX switch agent implements this function by sending a standard statistics request message to the switch's OpenFlow data plane, and then returning the results to the module. This

function allows an OFX module to count the number of bytes and packets in each flow without requiring packets to leave the data plane.

Finally, the `SendPacketIn` function sends a packet to the controller encapsulated in a standard OpenFlow packet_in message. The packet in message will *not* be received by the OFX module's controller handler, but rather by the control application's OpenFlow *packet_in* handler. This function allows OFX modules to process and filter packets before the switch sends them to the controller's standard OpenFlow handler.

B. The OFX Library

The OFX library implements the OFX controller API that allows an OpenFlow control application to interface with the OFX system. It provides two functions, summarized in Table II. First, the `LoadModule` function that loads an OFX module into the control plane. This function assigns the module a unique ID and returns a reference its *controller interface*. Second, the `PushModule` function, which installs a loaded module onto a switch in the network. This function encapsulates the module code inside of an OpenFlow experimenter message and sends it to the specified switch. The OFX switch agent running on that switch receives the message and loads the switch components of the module.

OFX Messages

The OFX library also defines messages that allow the controller and switch components of OFX and any loaded OFX modules to communicate. OFX messages are encapsulated inside of OpenFlow Experimenter Messages [7], and transmitted on the standard OpenFlow connection that a controller maintains with its switches. Experimenter messages provide a standard way for OpenFlow switches and controllers to implement additional features within the OpenFlow message type space. An Experimenter message has two header fields: experimenter ID and experimenter Type. OFX sets the ID field to a predefined constant, so that the OFX library and OFX switch agent can distinguish OFX messages from other experimenter messages. OFX sets the type field equal to the unique ID of the module sending the message. OFX reserves a type for *OFX system messages*, which are the messages that the controller and switch send to each other to manage OFX agents (*e.g.* the message that sends a module from the controller to a switch). The maximum length of an OpenFlow message is 65535 bytes. Besides this constraint, the format of the remaining message is defined by the module. In our modules, each message begins with a 32 bit length field, followed by a 32 bit message type field, followed by the message contents.

C. The OFX Switch Agent

The OFX switch agent proxies the OpenFlow connection between the switch's OpenFlow management agent and the controller, which allows it to intercept OFX messages as defined above. The OFX switch agent implements most of the OFX module API functions, loads modules onto the switch, and manages the switch's OpenFlow tables to direct packets to OFX module packet handlers in the OFX data plane agent.

Loading Modules

When the OFX switch agent receives a message from the controller that instructs it to load a module, it:

- Writes the module's code, which is included in the message, to a local temporary file.
- Loads the module's switch handler function, and registers it in a local table so that it gets called whenever the OFX switch agent receives an experimenter message with the module's ID in its type field.
- Compiles the module's packet handler, which is implemented in C, and dynamically links it to the OFX data path agent.
- Executes the module's startup function.

Flow Table Management

The OFX switch agent implements the API functions that direct packets to a module's packet handler by installing flow rules into OFX managed tables on the switch's forwarding engine (*e.g.* OpenFlow compatible ASIC hardware). When the OFX switch agent starts, it initializes several OFX managed OpenFlow tables that match packets *before* they reach the controller managed forwarding table. Then, when an OFX module requests that its packet handler receive packets matching a specific pattern (using the OFX module API), OFX installs rules into the OFX managed tables to direct those packets to the OFX data plane agent, where the module's packet handler processes them. By installing these rules to a separate table, OFX prevents security modules from interfering with the switch's standard forwarding table.

The OFX switch agent configures a pipeline of four flow tables, as Figure 5 depicts. When a switch running OFX receives a new packet, it first matches the packet against the OFX intercept flow tables (*i.e.* the tables that OFX adds rules to when a module calls `AddInterceptFlow` or `AddNegativeInterceptFlow`). These tables implement the following logic:

- If the packet matches a flow rule on the negative table, it bypasses the positive table and continues to the tap tables.
- If the packet matches a flow rule on the positive table, it is forwarded to the OFX data plane agent, which will process the packet and then return it to the main flow table.
- If the packet does not match a flow rule on either table, it continues to the tap tables.

Once a packet passes the intercept tables, it is guaranteed to reach the main flow table without leaving the forwarding engine. The tap tables implement similar logic to decide whether or not to send a *copy* of the packet to the data plane agent:

- If the packet matches a flow rule on the negative table, it bypasses the positive table and continues to main flow table.
- If the packet matches a flow rule on the positive table, a copy of the packet is forwarded to the OFX data

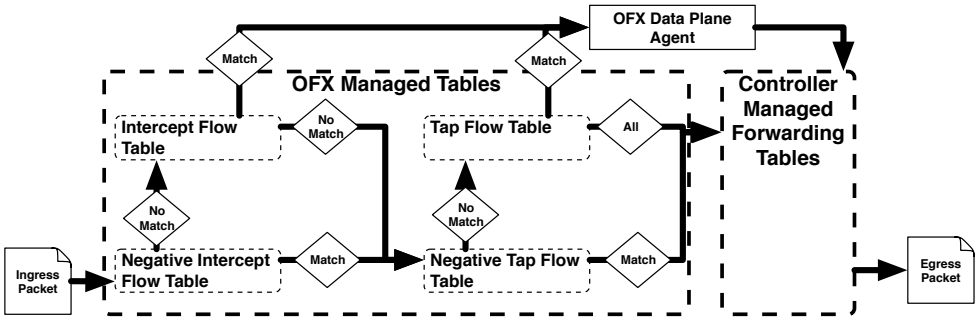


Fig. 5: OFX allows modules to specify which packets they want to intercept or tap using match patterns that OFX installs into a pipeline of forwarding tables that check packets before they reach a switch's standard forwarding tables.

plane agent, while the existing packet is continues to the main flow table.

- If the packet does not match a flow rule on either table, it continues to main flow tables.

D. Data Plane Agent

The data plane agent copies a packet from the data plane of the local switch into a buffer in the switch's system memory, and then calls the packet handlers of any modules that registered to intercept or tap matching packets, passing the handlers a pointer to the buffer. To determine which packet handlers to call, the data path agent keeps a local copy of all the module flow tables. If multiple modules registered to receive the packet, the data path agent calls them in order of the modules' priority, a value that the OpenFlow control application can set when it loads a module.

After calling all of the packet handlers, the data plane agent either sends the packet to the switch's main forwarding table, or drops the packet. Two conditions can cause the data plane agent to drop a packet: first, the packet handler of an agent registered to intercept the packet returned a code specifying that the packet should be dropped; second, all of the modules that registered to handle the packet did so in tap mode, in which case the OFX pipeline already sent a copy of the packet to the main forwarding table.

E. Implementation Details

Our current prototype of OFX is implemented to work with the Ryu control platform [6], and any OpenFlow switches running Linux. The OFX library is implemented as a Python module that any Ryu control application can load. The OFX switch and data plane agents are implemented as daemons that run on a switch, in Python and C respectively. OFX modules are written in Python, with the exception of the packet handler, which is a C function.

On the Pica 8 3290 that we used for our experiments in Section VI, we connected the OFX data plane agent to the switch's forwarding engine using a raw socket bound to a spare management port that was physically connected to a port on the forwarding engine. We took this approach because the low level kernel interface to the switch's Broadcom forwarding engine was very inefficient and completely closed (*i.e.*, the forwarding engine's firmware, driver, and even driver API were closed source). Recently, companies including Broadcom have begun initiatives to open up more of their API [1], [2], which can lead to an improved OFX interface. OFX can also play a role in motivating future switch designs with more open and higher capacity interfaces to the forwarding engine.

V. OFX Security Modules

In this Section, we describe three OFX security modules that we have implemented. These modules are motivated by: 1) the functionality that Avant-Guard [35] proposed as a data plane extension in a custom switch design; 2) the example declassifier and botnet detector applications discussed in Section II.

A. An Avant-Guard Inspired Module

This module provides OpenFlow control applications with two additional functions that can be loaded onto the switches in their networks: *trigger alerts*, which instruct a switch to send an alert to the controller if the packet or byte rate of traffic destined for a monitored host exceeds a certain value; and *TCP handshake validation*, which allows a switch to send a message to the controller when it detects that a new TCP connection has completed the initial three way handshake.

Avant-Guard extended the data plane of a software OpenFlow switch to achieve similar functionality. Our OFX implementation has a major deployment advantage, since it can be installed onto any OpenFlow switch that runs OFX, even ones with forwarding engines implemented in hardware. Further, it does not require modifying the switch's OpenFlow software or the controller platform used to manage the switches.

Trigger Alerts This functionality allows a switch to signal its controller whenever the packet- or byte-rate of the traffic destined for a host on the network exceeds a certain threshold. When a trigger fires, it sends the controller statistics about the packet and byte counts of *all* the recent flows that have connected to the monitored host. To achieve this functionality with standard OpenFlow, a controller would have to add a flow rule for each TCP or UDP connection, and then poll the switch for the statistics of the installed rules. In Section VI, we find that by using trigger alerts instead of polling, an OpenFlow control application can detect DDoS attacks much quicker.

Using Trigger Alerts Once loaded, the module exposes two trigger related functions to the control application. First, the `InstallTriggerAlert(IpAddress,`

`type <bytes or packets>, rate, switchID, updatePeriod)` function, which the control application can call at any time to install a trigger. Second, the `RegisterAlertHandler(FcnPointer)` function, which the control application can use to register a function to handle trigger alerts when they arrive at the controller.

Trigger Alert Implementation Details

- The `InstallTriggerAlert` function packs the input provided by the control application into an **install trigger message** that is defined in the module, and calls the OFX function `SendMessageToSwitch` to send the message to the switch that the control application wants to act as a monitor.
- When the message arrives at the switch, the OFX switch agent calls the module's switch handler with the message as input. The switch handler calls `AddTapFlow` (from the OFX API) which registers the module's packet handler to receive a copy of each packet destined for the host specified in the install trigger message. The handler also adds an entry to a trigger information table that stores the packet and byte counts values of each flow involving the monitored host. Each time the module's packet handling function receives a packet, it updates the table. A *trigger monitoring thread* that starts when the module is installed on the switch checks the entries in the table to determine if any alerts need to be send to the controller.
- Whenever the module's packet handler receives a packet destined for an a monitored host, it extracts the packet's flow key and updates the entry in the trigger information table for that flow. If no entry exists for the flow, it adds one. The module's packet handler then calls the OFX `AddNegativeTapFlow`, using the flow's key as the match pattern. This reduces the load on OFX by stopping the data plane from sending copies of future packets of that flow to the module's packet handler.
- The trigger monitoring thread periodically calls the OFX function `GetFlowStats` to get the byte and packet counts of all the added flows. When it receives the results, it updates the trigger information table to include the latest statistics from the OpenFlow data plane, checks to see if any of the trigger conditions have been reached for monitored hosts, and then sleeps for `updatePeriod` seconds before repeating. If it detects that a trigger condition has been reached for a host, it generates a **trigger alert message** that contains all the flow records involving that host, and then sends the message to the controller using the OFX function `SendMessageToController`.
- When the trigger alert message reaches the controller, the OFX library intercepts it and passes it to the module's controller handler. If the control application registered a trigger handler function, the module calls that function.

TCP Handshake Validation This functionality allows a switch to verify that a TCP flow has completed its handshake before sending a packet from the flow to the controller, which protects OpenFlow control applications that install routes for each TCP flow from Syn flood attacks. In a standard OpenFlow network, per TCP flow routing can only be implemented by sending the first packet of each new TCP connection to the controller via a *packet_in* message, which would then install a flow rule on the switch to handle future packets of that flow. The authors of Avant-Guard [35] found that this leaves the network vulnerable to Syn flood attacks, where an attacker floods the control plane and brings down the network by simply sending TCP syn packets with randomized sources, destinations, and ports. The originally proposed Avant-Guard solution to this vulnerability required modifications to a switch's forwarding engine, and therefore could not be deployed onto dedicated switches that have forwarding engines implemented in hardware. The TCP Handshake Validation component of our OFX Avant-Guard module is, to our knowledge, the first solution that can be used on such switches.

Using TCP Handshake Validation A control application can use this function as follows:

- First, the control application must call the `EnableHandshakeValidation(switchID)` function for any switch that should use the feature. The OFX Avant-Guard module exposes this function to the control application when it is loaded.
- Second, the control application must install default forwarding rules onto the switch to handle TCP packets that belong to unestablished connections.

After taking these two steps, the control application's standard packet_in handler, which contains the logic to install a new flow rule onto the switch to route each TCP flow, will only receive packet's from TCP connections that have completed their handshake.

TCP Handshake Validation Implementation Details

- The `EnableHandshakeValidation` function sends a message to the switch signaling it to enable TCP handshake validation, using the OFX `SendMessageToSwitch` function.
- When the message arrives at the switch, OFX passes the message to the module's switch handler, which registers the module's packet handler to intercept all TCP traffic, using the OFX `AddInterceptFlow` function.
- When the packet handler receives a TCP packet, it checks the connection status of the packet's flow using a hash table. If the packet belongs to a connection that is not yet fully established, the handler simply returns the packet to the data plane, where it will be forwarded to its destination by default flow rules that the controller installed in the main flow table. If the packet belongs to a connection that has completed the three way connection, the packet handler sends the packet to the controller using the OFX `SendPacketIn` function, so the controller can install the appropriate flow

rule into the main flow table to route this specific TCP flow. The packet handler then registers to *not* receive future packets belonging to this specific flow, using the OFX `AddNegativeInterceptFlow` function. To prevent sending multiple packets from a single flow to the controller, the packet handler keeps a hash table cache of all the recently added negative intercept flow rules. If the packet matches an entry in this table, the packet handler still returns the packet to the data plane, but does *not* send a copy to the controller.

B. A Traffic Declassifier Module

We implemented an OFX module that supports the taint-tracking declassifier application, originally discussed in Section II. When a control application loads this module, it installs the declassifier logic into switches in the network and opens a socket that other elements of the taint tracking system can use to send permissions to the switches.

Startup The traffic declassifier module only exposes one function to a network control application: `StartDeclassifier()`. This function sends a start message to the switches running the Silverline module. Upon receiving the message, each switch calls the OFX `AddInterceptFlow` function to register that all TCP packets should go through the declassifier module's packet handler. Finally, after loading the module on all the switches, the `StartDeclassifier` function on the controller starts a socket thread to forward permission information from the taint-tracking system to the switches.

Operation When the declassifier thread on the controller receives a permission update from the declassifier database, it packs the record into a **update permissions** message and sends it to each switch running the declassifier module.

When a switch receives a permission update message, the declassifier switch handler loads the permissions into a local hash table that map client IP addresses to lists of taint tags that clients from that IP address have permission to view.

Finally, whenever the declassifier packet handler receives a TCP packet, it extracts the ToS and destination address fields of the IP header. It looks up the destination (*i.e.* client) address in the permission hash table. If the client permission list includes the taint tag in the ToS field, the packet handler returns the packet to the main flow table of the data path, where it can be forwarded out, and calls the OFX `AddNegativeIntercept` function so that future packets from this flow, with this IP ToS field, bypass the packet handler. If the client permission list does *not* include the permission required to view the packet, the packet handler generates a TCP connection reset packet directed to the *server*, and returns that packet to the switch's forwarding table instead of the original packet directed to the client.

C. A Botnet Detector Module

We also implemented an OFX module to support the example botnet detection application. When a control application loads this module, it installs data collection logic onto switches that gathers flow records, opens a connection to an analysis process, and then periodically forwards flow records from the switches to the analysis process.

Startup The botnet detector module exposes one function to the controller: `StartBotDetector(UpdatePeriod)`, which starts the data collection module on all the switches where it has been loaded. The UpdatePeriod parameter specifies how frequently the switches should send data updates to the controller. The `StartBotDetector` function sends a message to the network's switches that instructs them to begin collecting data. When a switch receives the message, it runs a start up function that registers the modules packet handler to initially receive a copy of *all* packets that pass through the switch, using the OFX function `AddTapFlow`. The module's switch start up function also initializes a hash table that collects two the packet and byte count of each TCP or UDP flow and starts a background process that periodically collects the flow data and sends it to the controller in a **data update messages** .

Operation When the packet handler on a switch receives a packet, it extracts the packet's layer 3 flow key (*i.e.* source, destination, source port, destination port, IP protocol type). It then checks the local flow hash table to see if an entry exists for the flow. If an entry does not exist, it creates one and records the flow's packet and byte counts to the entry. If the entry does exist, it updates the entry's counters. Finally, the packet handler installs a negative tap rule using `AddNegativeTapFlow`.

The update thread on the switch execute the following loop:

1) Get the statistics of all the flows that the packet handler installed using `GetFlowStats`.
2) Sum the returned results with the counts in the local hash table.
3) Send the summed records to the controller.
4) Sleep for `UpdatePeriod`.

When the controller receives an update from a switch, it forwards the data to the analysis process.

VI. Evaluation

With OFX, security applications can leverage data plane programmability to enhance their operation and improve performance. In this Section, we benchmark OFX on a testbed with a hardware OpenFlow switch, and focus on answering the following questions:

1) What is the raw processing overhead of using OFX?
2) Can we get the performance gains Avant-Guard achieved with custom data plane extensions, but with greater deployability using OFX's programmable framework?
3) How much of a performance gain can a security application achieve by using OFX, when compared to traditional middlebox deployments and more recent SDN based security applications?

A. The Testbed

Figure 6 illustrates the testbed network for our experiments. It contained a network infrastructure consisting of: a hardware

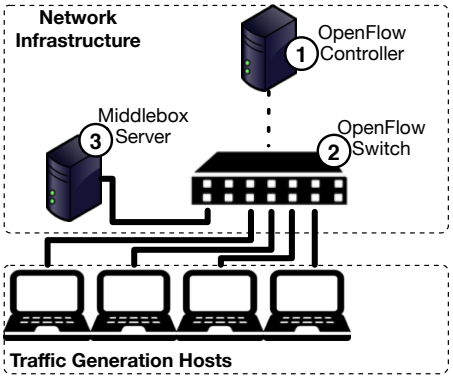


Fig. 6: A diagram of our testbed network.

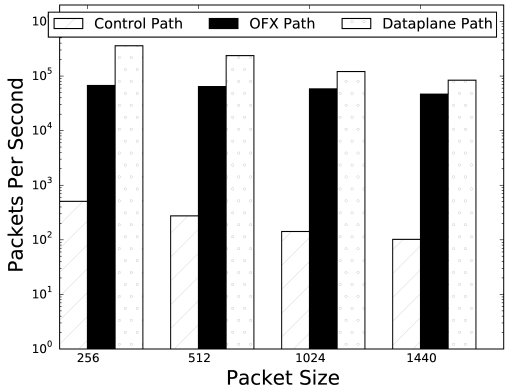


Fig. 7: Packet size vs packets per second using different paths through our testbed network.

OpenFlow switch, a Pica8 3290 with a Broadcom Firebolt-3 forwarding engine that processes packets in hardware according to OpenFlow rules and a 825 Mhz PowerPC CPU with 512MB of memory that runs Debian 7; a control server, a quad-core Intel i7 machine with 4GB of RAM, running Ubuntu 14.04 server LTS and an OpenFlow control platform (either the reference Open vSwitch reference controller with version 2.3, or the Ryu controller [6]); and a middlebox server, also an Intel i7 machine with 4GB of RAM, running Ubuntu 14.04 server LTS. The controller connected to the first management port of the switch with a Gigabit NIC, and the middlebox connected to a port on the switch's data plane, also using a Gigabit NIC. We ran traffic through the testbed network using up to four traffic generation machines, which were dual-core Intel Core-2-Duo machines with 2GB RAM. Each traffic generator was connected to a separate port on the Pica8 switch's data plane with a Gigabit NIC.

B. Raw OFX Overhead

To benchmark the raw overhead of using OFX, we measured its performance in a scenario where we directed *every* packet through an OFX packet handler that copies each packet into system memory, performs no operations to the packet, and then sends the packet back out to the data plane. We compared

Statistic	Control Path	OFX Path	Data Path
Min RTT	3.604 ms	0.251 ms	0.169 ms
Avg RTT	4.039 ms	0.31 ms	0.232 ms
Max RTT	8.08 ms	0.405 ms	0.292 ms
Max TCP Throughput	1.2 Mbps	584 Mbps	847 Mbps
UDP Drop % @ 5MBPS	72 %	0 %	0%
UDP Drop % @ 50MBPS	-	0.13 %	0%
UDP Drop % @ 500MBPS	-	3.6%	0%

TABLE III: Traffic statistics for flows travelling through different paths on our testbed.

with two baselines, a path through the network where the packets stayed entirely in the switch's hardware forwarding engine, and a path through the network where the packets went through the controller. For these benchmarks, we ran the Open vSwitch reference controller on the control server, and used two traffic generation machines. Figure 7 shows the maximum number of ping packets one traffic generation host can send to the other using the hping3 [9] tool and Table III summarizes other network statistics measured using Iperf [36]. For all measurements, the OFX path performed several orders of magnitude better than the control path. The main bottleneck for both the control path and OFX path was the CPU usage of the switch, though OFX was much more efficient, which allowed it to perform better with respect to every metric.

C. Deployable Data plane Extensions

Next, we benchmarked the Avant-Guard inspired module by comparing the performance of Ryu based OpenFlow control applications that used the module, with ones that implemented equivalent functionality without using the module.

TCP Handshake Validation To benchmark TCP Handshake Validation, we implemented a simple TCP learning OpenFlow control application that installs a flow on the switch whenever it receives a packet from a previously unseen TCP stream. We then compared the application to a slightly modified version that loads the Avant-Guard inspired module and uses the TCP Handshake Validation function, which allows the switch to only notify the controller about a new TCP flow *after* the TCP handshake has completed [1]. We connected an attack host to our testbed that sent a flood of TCP syn packets to the controller, and then measured the likelihood of two other hosts establishing a TCP connection within 6 seconds.

Figure 8 shows the percentage of 20 TCP connection attempts that succeeded, for trials in which the flood rate varied. Without the handshake validator, the attack packets quickly overloaded the switch's ability to send packets to the controller, preventing the legitimate connections from being established. Running the OFX module allowed the network to be resistant to attacks several orders of magnitude larger. In the original Avant-Guard proposal [35] , the authors demonstrated that their custom data plane achieved similar performance

[1]Before a flow's TCP handshake is completed, the switch forwards the packets according to a default strategy, we used simple rules that matched the Ethernet destination address.

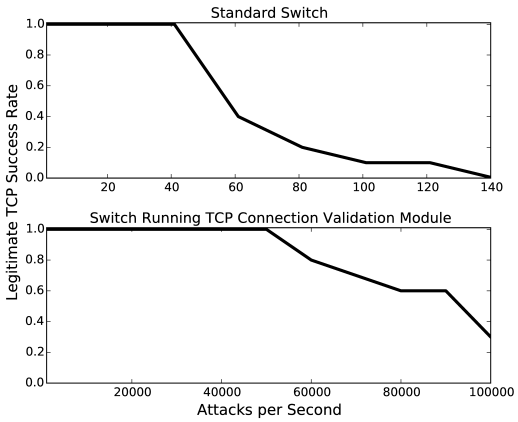


Fig. 8: Attack packets per second vs TCP connection success rate, for a standard OpenFlow network, and a network using the OFX TCP Connection Validation function.

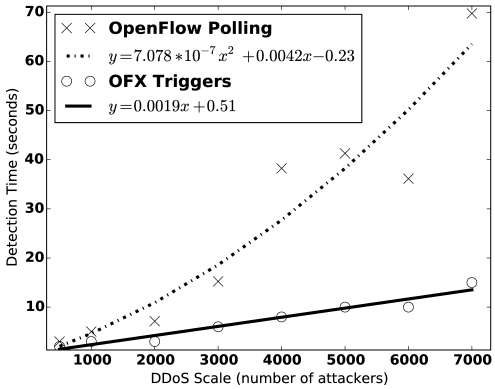


Fig. 9: The amount of time it took a control application to detect a 600 Mbps DDoS attack from a varying number of sources, while using OpenFlow polling or trigger alerts from the OFX Avant-Guard inspired module.

benefits [2] However, the Avant-Guard data plane cannot be deployed onto existing OpenFlow switches, whereas the OFX module can add this feature to *any* OpenFlow switch, even those with hardware data planes.

Trigger Alerts To benchmark trigger alerts, we wrote a control application that monitors the bandwidth usage of each traffic flow connecting to a host in the network, and generates an alert when the total bandwidth usage exceeds a threshold value. We compared two implementations of the control application: a standard OpenFlow version, that installs a flow rule for each new IP source address and then monitors for DDoS attacks by periodically polling the switch for the statistics of all the rules; and an OFX version that installs a trigger alert that which collects the flow statistics on the switch and only sends the statistics to the controller when the trigger

[2]In the Avant-Guard proposal [35], figure 13 showed that validating TCP handshakes with a custom data plane provided resistance to attacks at least 1 order of magnitude larger, but did not display larger results.

Workload Name	Median	High Bandwidth	Frequent Arrival
Flow Inter-arrival Period	.015	.15	.0015
Per Flow Packet Rate	250	500	25
Flow Duration	1	30	.1
Packet Size	300	1400	100
Number of Flows	1000	100	10000
Average Transmission Bandwidth	43.57	970.99	19.75

TABLE IV: Workloads used to benchmark the declassifier and botnet detector modules.

condition is reached. In both implementations, the update rate was set to 1 second, and the threshold for generating an alert was 500 Mbps. We generated TCP flows that simulated a 600 Mbps DDoS attack from varying numbers of source, and sent the flows from one traffic generation host to another, through the network.

Figure 9 shows the amount of time it took the two implementations to detect that a DDoS attack of greater than 500 Mbps was occuring. Both versions took longer to detect DDoS attacks that were more distributed because they had to install a counting rule for each source IP address that connected to the monitored host. However, the trigger alert function allowed the switch to install the rules without forwarding packets to the controller, which led to them being installed much quicker and greatly reduced the detection time.

D. OFX Security Applications

Finally, to understand how OFX can enhance the operation of existing security applications, we benchmarked OFX as a platform for deploying the taint tracking declassifier and botnet detection applications that we used as running examples throughout the paper. We compared three version of these applications:

- A *standard OpenFlow control application*, as described in Section II, that runs on the network's OpenFlow control server (**1** in Figure 6).
- A *middlebox application* that runs on a dedicated server in the network (**3** in Figure 6). Our implementations were functionally equivalent to the OpenFlow and OFX versions, and written in C. The declassifier application ran *inline* (*i.e.* we configured the switch route each packet *through* the declassifier before going to its destination). The botnet detection application ran as a *tap* application (*i.e.* we configured the switch to send a *copy* of each packet to the detector.)
- An *OFX enhanced application* that runs on the controller, but loads and deploys the application specific modules described in V onto the switch. With this deployment strategy, the security application is distributed across both the controller and switches in a network (*i.e.* **1** and **2** in the diagram of our testbed, Figure 6.)

We ran three traffic workloads through the applications, which Table IV summarizes. The *median workload* models median traffic conditions observed in previous studies. A large scale data center study [26] reported that the median flow inter-arrival period for a top of rack switch, similar to the switch in our testbed, was .015 seconds. The other statistics

were median TCP flow statistics reported in a large scale TCP flow analysis [33]. The other two workloads represented conditions that stress different aspects of the systems. In the *high bandwidth workload*, flows last longer and use significantly more bandwidth, but there are fewer flows and they arrive an order of magnitude less frequently. This workload stresses the systems' ability to process data at high rates. The *frequent arrival workload* contains more flows that arrive an order of magnitude quicker than those in the median workload, but use much less bandwidth individually. This workload stresses the systems' ability to manage a large number of quickly arriving flows.

We used four metrics to compare the implementations:

- **Latency**, or how long it takes a packet to arrive at its destination, once it leaves its source.
- **Time to First Packet**, or the latency of the first packet of each flow. This metric measures flow setup time.
- **Observed Throughput**, or the throughput observed by the hosts using the network.
- **Security System Drop Rate**, or the percentage of packets that a security application dropped.

Taint Tracking Declassifier Figure 10 shows the cumulative distribution of how much delay the declassifiers added to packets in each workload. The OFX version added very little delay to most packets because after checking the permission of the first packet of a flow, it installed a rule into the data plane to handle the remaining packets of that flow without forwarding them to the switch's CPU. The OpenFlow version also used this approach, and performed well in the high bandwidth workload. However, latency increased in the median and frequent arrival workloads because the flows arrived more quickly than the OpenFlow declassifier could install rules, so many packets ended up queued in the slow path to the controller. The middlebox deployment could not add flow rules to the data plane, and so it had to process each packet. This worked well in the frequent flow arrival and median workloads, but in the high bandwidth workload, the packet rate was too high and the middlebox application ended up queuing approximately 30% of the packets, adding several orders of magnitude more delay to them.

Figure 11 shows the cumulative distribution of how much delay the declassifiers added to *the first packet of each flow*. the OFX version added latency to these packets because they went through the OFX module's permission checking packet handler. For 90% of the flows, the OFX version added under 2ms of latency to the first packet, across all workloads. This was approximately 1 order of magnitude more latency than it added to packets on average. However, for about 10% of the flows, the OFX version added up to 100 ms of latency. The bottleneck was the switch's weak single-core CPU: to install an OpenFlow rule into the data plane, the switch must context switch to a Broadcom driver process, which delayed the processing of whatever packet was being handled when the interrupt occurred. Many switches, especially newer models, would not have this bottleneck because they have faster CPUs with two or more cores.

Implementation	Frequent arrivals	Median	High Bandwidth
Middlebox	19.74	43.55	733.13
OFX	18.75	42.18	970.90
Controller	0.50	5.16	970.43
Maximum Possible	19.75	43.57	970.99

TABLE V: Average network throughput (in Mbps) while using the declassifier applications with different workloads.

Implementation	Frequent arrivals	Median	High Bandwidth
Middlebox	0.03%	0.04%	24.50%
OFX	5.01%	2.79%	0.01%
Controller	96.47%	75.91%	1.73%

TABLE VI: Declassifier drop rates during different workloads.

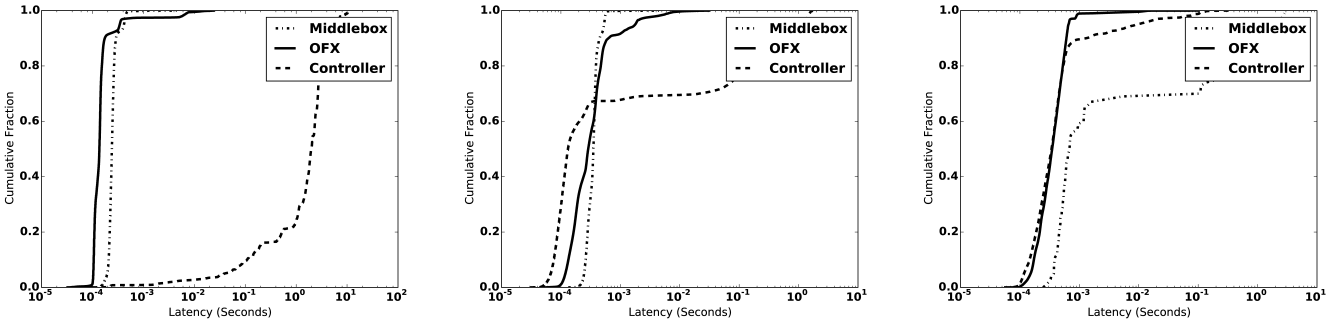
In comparison, the non OFX versions did not perform as consistently across the scenarios. The OpenFlow version performed poorly in all scenarios, because it required the first packet of each flow to travel the slow path to the control plane. The delay increased with flow inter-arrival rate because the path between the data plane and control plane is not only slow, but also has very limited throughput. During the median workload, the OpenFlow version added more than 10 ms of delay to the first packet of most flows, and more than than 1 second of delay for 10% of the flows. The middlebox distributions for time to first packet were the same as the middlebox distributions for packet latency because the middlebox processed every packet the same way.

Table VI summarizes the percentage of packets dropped by each declassifier, and Table V summarizes the throughput observed on the network while the declassifiers were in use. The OFX declassifier performed well in all scenarios. In the high bandwidth scenario, it installed flow rules quickly enough to avoid becoming overloaded by the high bandwidth flows. While it was not able to install rules for each flow in the other workloads with more quickly arriving flows, OFX's low level connection to the data plane provided enough bandwidth to handle most packets in software.

The other versions of the declassifier did not perform as consistently. The middlebox deployment did well in the lower bandwidth workloads. However, the higher bandwidth workload saturated its 1 Gbps network interface and cause it to drop approximately 25% of the packets, and reduce the observed throughput . The controller deployment did well with the high bandwidth workload because it had enough time to install a rule into the data plane for each of the infrequently arriving flows. However, in the other workloads where flows arrived more quickly, the controller was not able to install a rule for each flow. Packets from the flows that it missed flooded the low bandwidth channel between the switch and controller, and many ended up dropped.

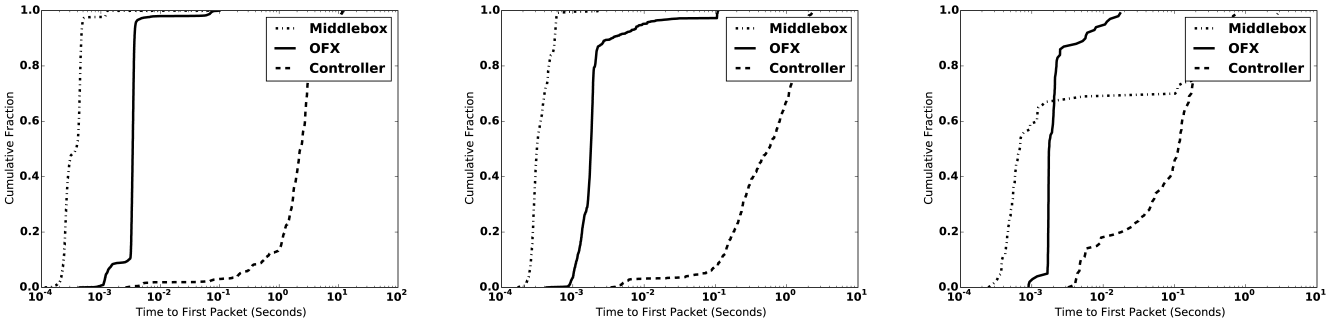
Botnet Detection Unlike the taint tracking declassifier application, which must run *inline* (*i.e.* on the path between a protected server and a client), the botnet detection application runs as a *tap* based application (*i.e.* it receives a copy of each packet, but is not directly on the path between two hosts).

Figure 12 shows the latency the different versions of this application added to packets in the median workload, and



(a) Frequent arrival workload (b) Median workload (c) High bandwidth workload

Fig. 10: CDF of packet latency for declassifier applications with each workload.



(a) Frequent arrival workload (b) Median workload (c) High bandwidth workload

Fig. 11: CDF of time to first packet for the declassifier applications with each workload.

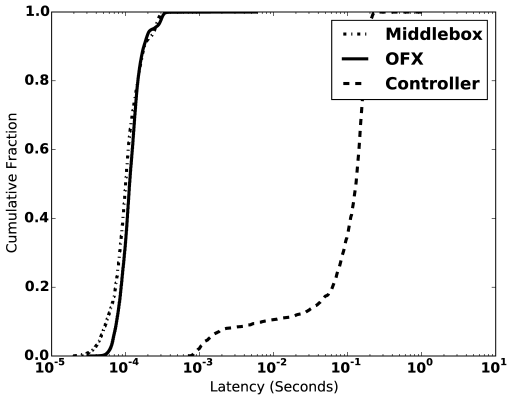


Fig. 12: CDF of packet latency for botnet detector deployments during the median workload.

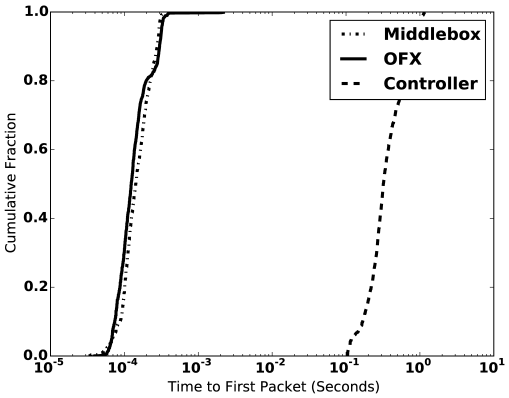


Fig. 13: CDF of time to first packet for botnet detector deployments during the median workload.

Figure 13 shows the latency added to the first packet of each flow. The middlebox and OFX deployments added minimal latency to packets, regardless of whether or not they were the first packet of a flow, because the switch's forwarding engine can copy packets very quickly (*i.e.* approximately 1 microsecond according to the vendor [4]). The OpenFlow version added much more latency because the switch had to forward a copy of the packet to the controller before sending the original packet out. This affected the first packet of each flow more, since the OpenFlow implementation installs a counting rule to handle the remaining packets in a flow as soon as the controller receives a packet from the flow. The time to

Implementation	Frequent arrivals	Median	High Bandwidth
Middlebox	19.75	43.57	970.99
OFX	19.75	43.57	970.99
Controller	2.68	5.90	967.09
Maximum Possible	19.75	43.57	970.99

TABLE VII: Observed network throughput (in Mbps) while using the botnet detection application during different workloads.

Implementation	Frequent arrivals	Median	High Bandwidth
Middlebox	0.07%	0.03%	21.67%
OFX	7.09%	5.09%	0.47%
Controller	94.92%	80.86%	0.89%

TABLE VIII: Botnet detection application drop rates during different workloads.

first packet and latency measurements for the other workloads show the same trends.

Table VII summarizes the throughput observed in the network while using the botnet detection applications with different workloads. As was the case for network latency, the OFX and middlebox versions did not affect the network throughput at all, due to how quickly the data plane can copy packets. The controller version affected throughput, especially for the workloads with higher flow inter-arrival rates, since it could not install rules into the data plane fast enough to prevent the switch from executing the slow *copy to controller* operation on a large number of packets.

Table VIII summarizes the drop rate of the botnet detection applications themselves. Based on this metric, the OFX version performed the most consistently, dropping at most 7.09% of packets in the frequent arrival scenario. There were two factors that contributed to this drop rate: first, the switch could not install rules into the data plane quickly enough to prevent many of the packets from being sent to the OFX software based packet handler; second, the switch CPU was weak, which caused it to drop a fraction of those packets.

The middlebox version performed significantly worse during the high bandwidth scenario, droppping 24% of the packets. This is a limitation of the middlebox's 1 Gbps NIC, which was not able to keep up with the high bandwidth. The control plane version of the application performed poorly in all scenarios, again due to the limited throughput of the path between the data and control planes.

Security Application Results Summary Overall, these results support the conclusion that OFX is a a better platform for network security applications than either of the two alternatives we tested.

- The OFX enhanced security applications performed the most consistently across all the workloads and metrics.
- The OpenFlow versions performed so poorly in most scenarios that they would be unusuable in practice. OFX provided a $20x - 40x$ capacity increase over these OpenFlow versions, based on Tables VI and VIII.
- The Middlebox versions performed adequately in most scenarios, but added *more latency* to most packets and did not scale to higher bandwidth scenarios as well when compared to the OFX versions. Based on Tables VI and VIII, OFX provided an approximately $1.25x$ capacity increase compared to the middlebox versions, in the high bandwidth scenario.

VII. Related Work

We are motivated by recent work that proposes changes to the OpenFlow data plane to support the needs of security and monitoring applications [16], [31], [35], [37]. These proposals are all for limited functionalities that only support specific applications. In contrast to these proposals, OFX provides a *programmable* framework that applications can use to define and load whatever functionality they need into their switches' data planes. There is a large body of work on building extensible data planes, including virtual switches such as Open vSwitch [11] and Click [27], frameworks such as SwitchBlade [13] for implementing customized data planes on configurable hardware (*i.e.*, FPGAs), proposals to build network switches out of general purpose servers [17], and programming models for the protocol parsing hardware of switches [14]. OFX takes an alternate approach: it allows users to extend the *non-modifiable hardware* data planes of *commodity network switches* with *software* based functionality.

We also contrast OFX with OpenFlow control platforms that seek to improve performance. Onix [28] and Kandoo [23] allow network operators to build control applications that run across multiple servers and Beacon [18] supports multi-process control applications. OFX also allows parts of a control application to be distributed, but across switches instead of servers or CPU cores. By running on the switch, OFX allows applications to avoid sending packets through the OpenFlow control channel, which can itself be a significant bottleneck. OFX also lets the module define how it distributes work across the switches, which may provide more opportunities for application specific optimization compared to other general approaches to control plane distribution.

OFX draws inspiration from previous control platforms that seek to simplify the development of SDN applications, such as FRESCO [34], a network control platform for building and deploying module-based control applications, and Frenetic [21], a high level language for writing network control programs. The programming models in these systems could be applied to OFX, which is a framework for building and deploying switch extension modules instead of network control applications. An older but closely related area is active networking, such as Active Bridging [12], which introduced the concept of dynamically extending the functionality of network nodes. Active networking nodes were implemented on general purpose servers, and had no interface to a centralized controller because the techniques predated the concepts of data and control planes. The Tiny Packet Programs [25] framework revisited this idea and proposed new switch hardware that can execute simple code embedded in packets. In comparison, OFX allows existing, unmodified switches to execute more complex code that can be installed by the controller.

VIII. Conclusions and Future Work

Software Defined Networking has much to offer security applications. However, current approaches to building SDN based security applications are not practical because of performance limitations and deployment hurdles. The OFX framework provides a better approach that overcomes these challenges. It allows security applications to improve performance by extending switches with custom functionality and works within existing OpenFlow infrastructures. Our sample modules and evaluation demonstrate how OFX can improve the performance of representative security applications running on real OpenFlow hardware and software, in a variety of scenarios. OFX is a first step towards allowing control applications to dynamically install custom software based functionality into the data plane. There are many future directions for research that build on this idea, including: further optimizing the interface between the switching hardware and the OFX agents; refactoring more applications to use OFX; testing OFX on other hardware platforms; and exploring alternative programming models for building OFX modules.

Acknowledgements: We wish to thank the anonymous reviewers for their input on this paper. This research was partially supported by NSF SaTC grant numbers 1406192, 1406225, and 1406177.

References

[1] Broadcom of-dpa overview. http://www.broadcom.com/collateral/pb/OF-DPA-PB100-R.pdf.

[2] Broadcom open nsl. https://github.com/Broadcom-Switch/OpenNSL.

[3] "Cisco IOS Embedded Event Manager (EEM)," http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-embedded-event-manager-eem/.

[4] "Pica 8 p3290 data sheet," http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf. [Online]. Available: http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf

[5] Pica os white box switch os. http://www.pica8.com/white-box-switches/white-box-switch-os.php.

[6] "Ryu," http://osrg.github.io/ryu/.

[7] (2012) Openflow switch specification v1. 3.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.2.pdf. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.2.pdf

[8] "Floodlight," http://floodlight.openflowhub.org, 2013.

[9] "Hping 3," http://www.hping.org/hping3.html, 2014.

[10] "Open network linux," http://opennetlinux.org, 2014.

[11] "Open vswitch," http://openvswitch.org, 2014.

[12] D. S. Alexander, M. Shaw, S. M. Nettles, and J. M. Smith, "Active Bridging," in *Proc. ACM SIGCOMM*, 1997.

[13] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster, "SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware," in *Proc. ACM SIGCOMM*, 2010.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, July 2014.

[15] R. Braga, E. Mota, and A. Passito, "Lightweight ddos flooding attack detection using nox/openflow," in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*. IEEE, 2010, pp. 408–415.

[16] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks," in *Proc. SIGCOMM*, 2011.

[17] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: exploiting parallelism to scale software routers," in *Proceedings of the 22nd ACM SIGOPS*. ACM, 2009, pp. 15–28.

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

[19] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An api for application control of sdns," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 327–338.

[20] G. FERRO, "Big Switch Networks Launches Mature Hardware-Centric Data Centre SDN Solution," http://etherealmind.com/big-switch-networks-launches-hardware-centre-mature-data-centre-sdn-solution/, July 2014.

[21] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2011.

[22] G. Gu, R. Perdisci, J. Zhang, W. Lee *et al.*, "Botminer: clustering analysis of network traffic for protocol-and structure-independent botnet detection," in *Proceedings of the 17th conference on Security symposium*, 2008, pp. 139–154.

[23] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[24] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: transparent moving target defense using software defined networking," in *Proc. Workshop on Hot topics in software defined networks (HotSDN)*, 2012.

[25] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazières, "Tiny packet programs for low-latency network control and monitoring," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 8.

[26] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009, pp. 202–208.

[27] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug 2000.

[28] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *OSDI*, vol. 10, 2010, pp. 1–6.

[29] D. Medhi, *Network Routing: Algorithms, Protocols, and Architectures*, 1st ed. Morgan Kaufmann, 2007.

[30] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting traffic anomaly detection using software defined networking," in *Recent Advances in Intrusion Detection*. Springer, 2011, pp. 161–180.

[31] J. C. Mogul and P. Congdon, "Hey, You Darned Counters!: Get off My ASIC!" in *Proc. Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[32] Y. Mundada, A. Ramachandran, and N. Feamster, "Silverline: preventing data leaks from compromised web applications," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 329–338.

[33] L. Qian and B. E. Carpenter, "A flow-based performance analysis of tcp and tcp applications," in *Networks (ICON), 2012 18th IEEE International Conference on*. IEEE, 2012, pp. 41–45.

[34] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *Proc. Network and Distributed System Security Symposium (NDSS)*, February 2013.

[35] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-defined Networks," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[36] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf: The tcp/udp bandwidth measurement tool," http://dast.nlanr.net/Projects.

[37] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable Flow-based Networking with DIFANE," in *Proc. ACM SIGCOMM*, 2010.