Software-defined Networking: Improving Security for Enterprise and Home Networks

by

Curtis R. Taylor

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

May 2017

APPROVED:

Professor Craig A. Shue, Dissertation Advisor

Professor Craig E. Wills, Head of Department

Professor Mark Claypool, Committee Member

Professor Thomas Eisenbarth, Committee Member

Doctor Nathanael Paul, External Committee Member

Abstract

In enterprise networks, all aspects of the network, such as placement of security devices and performance, must be carefully considered. Even with forethought, networks operators are ultimately unaware of intra-subnet traffic. The inability to monitor intra-subnet traffic leads to blind spots in the network where compromised hosts have unfettered access to the network for spreading and reconnaissance. While network security middleboxes help to address compromises, they are limited in only seeing a subset of all network traffic that traverses routed infrastructure, which is where middleboxes are frequently deployed. Furthermore, traditional middleboxes are inherently limited to network-level information when making security decisions.

Software-defined networking (SDN) is a networking paradigm that allows logically centralized control of network switches and routers. SDN can help address visibility concerns while providing the benefits of a centralized network control platform, but traditional switch-based SDN leads to concerns of scalability and is ultimately limited in that only network-level information is available to the controller. This dissertation addresses these SDN limitations in the enterprise by pushing the SDN functionality to the end-hosts. In doing so, we address scalability concerns and provide network operators with better situational awareness by incorporating system-level and graphical user interface (GUI) context into network information handled by the controller. By incorporating host-context, our approach shows a modest 16% reduction in flows that can be processed each second compared to switch-based SDN.

In comparison to enterprise networks, residential networks are much more constrained. Residential networks are limited in that the operators typically lack the experience necessary to properly secure the network. As a result, devices on home networks are sometimes compromised and, unbeknownst to the home user, perform nefarious acts such as distributed denial of services (DDoS) attacks on the Internet. Even with operator expertise in residential networks, the network infrastructure is limited to a resource-constrained router that is not extensible.

Fortunately, SDN has the potential to increase security and network control in residential networks by outsourcing functionality to the cloud where third-party experts can provide proper support. In residential networks, this dissertation uses SDN along with cloud-based resources to introduce enterprise-grade network security solutions where previously infeasible. As part of our residential efforts, we build and evaluate device-agnostic security solutions that are able to better protect the increasing number of Internet of Things (IoT) devices. Our work also shows that the performance of outsourcing residential network control to the cloud is feasible for up to 90% of home networks in the United States.

Acknowledgements

First and foremost comes my wife, Mary Clare. She has supported me throughout my graduate studies. Without her support, I would have undoubtably not been able to complete this work. Additionally, I would like to thank my parents, who have also provided continual support throughout my entire life.

I owe my advisor, Craig Shue, an enormous thanks for all the time and effort spent in helping me develop as a researcher. I would also like to give thanks to the rest of my committee: Mark Claypool, Thomas Eisenbarth, and Nathanael Paul for their advice and feedback.

A special thanks to my fellow ALAS lab members over the years, including but not limited to: Hang Cai, Mohamed Najd, Douglas MacFarland, Joshua Pritchett, Daniel Moghimi, Salman Saghafi, Doran Smestad, Tim Nelson, and Evan Frenn. You all have been an invaluable resource and good friends. Finally, I would like to thank both the ALAS and PEDS research groups for listening and providing valuable feedback on my research. I am also grateful to have heard all the interesting research presentations in other research areas, which has helped broaden my horizon.

Material presented in Chapter 3 has been accepted for publication and is co-authored with Douglas MacFarland, Doran Smestad, and Craig A. Shue [166]. Material in Chapter 6 has been published in a peer-review venue and is co-authorized with Mohamed E. Najd and Craig A. Shue [168]. Similarly, material in Chapter 7 has been published in a peer-review venue and is co-authored with Craig A. Shue [167]. Finally, material in Chapter 4 is in preparation to be submitted to a peer-review venue and work in Chapter 8 has been submitted to a peer-review venue along with co-authors Tian Guo, Mohamed E. Najd, and Craig A. Shue.

This material is in part based upon work supported by the National Science Foundation under Grant No. 1422180 and the National Science Foundation Scholarship for Service Grant No. 1503742.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer networks are the transportation superhighway of modern computing by providing a means of interconnectivity between different physical and virtual platforms. These networks are expected to provide high performing and reliable data transport to and from different machines all over the world. In many ways, the push for higher performance to meet the growing demands of the Internet has left a gap in security. Operators are beginning to seriously consider what new approaches exist for securing their network. This dissertation work leverages relatively new networking techniques to provide security solutions that are impossible in modern enterprise and residential networking environments.

At a high-level, we see the evolution of the Internet having resulted in several key network infrastructure types. Of these types enterprise networks, cloud or data center networks, and residential networks are some the most frequently accessed. Among these three network types, there are many overlapping goals, including performance, security, privacy, between each network type, but the approach to achieving these goals is drastically different based on each network type's characteristics. Enterprise networks are networks that are largely self-sufficient and host many resources internally. Examples of enterprise networks include government, universities, and private companies. Similar to enterprises, public cloud infrastructure is self-sufficient in hosting resources. In fact, cloud providers often work in conjunction with enterprises by providing a general computing platform. Residential networks provide homes with Internet access and typically interact with services provided by enterprises. The number of residential networks far outweigh the number of enterprise connections at over 690 million worldwide [160].

Enterprise and residential networks directly host computing devices used by clients. In contrast, cloud infrastructure is typically used transparently by clients when enterprises outsource services, such as hosting webpages in the cloud. Enterprise and residential networks act as front-line networks by directly hosting clients and results in challenging security obstacles. These obstacles include providing scalable network security solutions that protect a diverse set of devices including desktops, mobile phones, and IoT devices.

The average American spends approximately two-thirds of their day between enterprise and residential networks [47]. As such, we refer to these networks as "front-line networks". If security compromises are to happen, they will likely involve either an enterprise or residential network. This dissertation considers the importance of front-line networks and how to provide better means of security to them. We present new methods to improve security in front-line networks using a networking paradigm called software-defined networking (SDN), which allows operators the ability to control a network from a logically centralized location. We do so by first rethinking how enterprise SDN is deployed. We then introduce SDN, along with cloud support, into the residential environment. The result of our work shows how SDN-based security solutions can be tailored to meet deployment challenges while considering each network's characteristics and limitations. In doing so, this dissertation addresses problems that exist in today's front-line networks and without our SDN techniques, would otherwise remain unaddressed.

We first consider how rethinking SDN deployment in the enterprise can address existing scalability and situational awareness concerns. Scalability and situational awareness become problematic due to network complexity. This complexity leads to situations where operators are blind to network traffic, making it difficult to have a complete, global view of the network. Network operators could mitigate risks such as malware spreading from a compromised system if they had a better understanding of the network activity from end-systems. In traditional systems, network operators are typically blind to intra-subnet traffic, since hosts directly forward the traffic without traversing security enforcement and monitoring devices. In an event of malware outbreak, containment techniques require shutting down important network services and even disconnecting the entire network to prevent the spread [161]. However, recent innovations, such as the software-defined networking (SDN) paradigm, hold the potential to partially address the problem. Using appropriately crafted fine-grained flows, the OpenFlow protocol [128], a popular SDN protocol, allows a centralized controller to learn each time a new network flow is created regardless of a host's position in the network.

The first major contribution of this dissertation is to apply host-based SDN techniques to the enterprise network. While switch-based SDN with OpenFlow may help address visibility concerns, such an approach faces two significant challenges: 1) fine-grained flows in OpenFlow's switch-based data plane do not scale to large networks [79] and 2) OpenFlow is inherently blind to end-host system activities, since it operates at the level of switches and routers. To answer these challenges, our work embraces the "dumb network, smart hosts" mindset. Specifically, we push OpenFlow functionality from switches and routers to end-host machines using a software agent. In doing so, we are able to address the aforementioned challenges. We distribute flow state to individual end-hosts helping reduce scalability concerns, and by running the agent on the end-hosts, we also gain insight into the host's operating context. With host-based SDN, we show how operators are able to write powerful, centralized security policies that include system level information such as process path in addition to incorporating user interactions via the graphical user interface (GUI) to endorse network traffic.

While there are many more residential networks than enterprise networks, most network security measures are directed at enterprises. Often, home networks are hosting devices that also frequent enterprise networks when "bring your own device" policies are in place, but when home, users operate under a vastly different network configuration. The difference is a result of enterprises having the resources to deploy strong security measures. Residential networks, on the other hand, are much more constrained in these aspects, which limits the viability of most protective measures.

Attempting to deploy even the most straightforward enterprise security measures can be challenging to execute in residential networks. Residential networks are different than the enterprise in multiple aspects. When compared to the enterprise, residential networks have vastly different network characteristics, limited infrastructure, IT experience, and cater to a heterogenous group of devices. This dissertation recognizes these differences and introduces an approach we refer to as *Residential SDN* (ReSDN). ReSDN is an attempt to build a solution that takes into account all of the previous considerations in order to provide a new residential network infrastructure supporting better network functionality. Our infrastructure is designed to be immediately deployable and practical from a performance perspective. ReSDN allows a third-party to remotely manage a home network using cloud-based infrastructure. To realize our infrastructure, ReSDN uses modified commodity home routers, which is the focal point of all residential network traffic. This dissertation shows the power of ReSDN and how future residential networks can be better managed and secured by performing the following research activities:

- **Outsourcing network functionality**: We begin addressing residential network limitations by outsourcing network functionality to the cloud. Rather than require the home router to provide network functionality locally, we configure the router to use cloud-based infrastructure to handle security and policy enforcement.

- **ReSDN application development**: We develop new network applications in the cloud that are impractical for traditional residential networks. These network functions include an automated privacy proxy for the popular Skype VoIP application called *SkyP* and a TLS verification and revocation application called *TLSDeputy.* These applications address real-world concerns for residential networks.

- **Large scale performance evaluation**: We leverage Amazon's Mechanical Turk [7] to perform a large scale performance evaluation of residential networks and public cloud providers. The performance evaluation shows that our approach is broadly applicable to approximately 90% of residential networks in the US.

- **Deployment of a residential testbed**: Using modified routers, we go through an IRB-approved study that enables us to deploy modified routers in real residential networks. This testbed enables us to gather data from multiple vantage points and in the future, will enable large scale evaluation of new services.

Chapter 2 presents background information that will provide an understanding of the foundation this dissertation work is built upon as well as providing relevant related work. Chapter 3 presents our host-based SDN approach related to our enterprise security efforts. Chapter 4 discusses how host context can be extended into the GUI. Chapter 5 introduces our ReSDN infrastructure while Chapters 6 and 7 discusses applications of ReSDN. Finally, Chapter 8 analyzes ReSDN performance on a large scale and Chapter 9 discusses our efforts towards building a ReSDN testbed.

Chapter 2

Background and Related Work

Software-defined networking (SDN), network function virtualization (NFV), and security are all important components of this dissertation. We recognize SDN's powerful abstraction, as well as its limitations. In some cases, we use it alongside NFV for implementing security components. We now provide an overview of SDN, NFV, and security with respect to our goals of security enterprise and residential networks. In other cases, we provide related work in each Chapter to provide better context.

2.1 Software-defined Networking

The following background relates to a relatively new networking paradigm known as software-defined networking. Software-defined networking is an approach that separates the data plane, or packet forwarding hardware, from the control plane, or the logic that decides if and how a packet should be forwarded. A defining feature of SDN is its ability to provide a *logically centralized* view of the network. In theory, this allows for improved security, performance, and resiliency in the network. Although multiple SDN implementations exist, academics and industry have chosen OpenFlow [129] as the *de facto* SDN standard. A high-level overview of the SDN abstraction is shown in Figure 2.1. For the remainder of this work, our SDN implementation and discussion will strictly consider the OpenFlow SDN implementation.

The logically centralized nature of SDN enables network management that would otherwise be impossible. As an example, consider a commodity unmanaged switch connecting machines at a university. Unfortunately, after these switches are deployed throughout the network, administrators do not have the ability control or alter the logic. Moreover, switches that do allow limited control often require physical access and a special adapter for configuration and are limited to the functionality provided by proprietary software. However, SDN decouples the control and data plane to allow consumers to use a manufacturer's hardware while having the ability to implement their own control software. The restrictions of modern switches has already been recognized by cloud providers and data centers. Accordingly, operators of these networks are using SDN to allow operators to quickly reconfigure the network or provision resources based on dynamic network conditions [117].

We now discuss the components that comprise SDN, which include the data plane, control

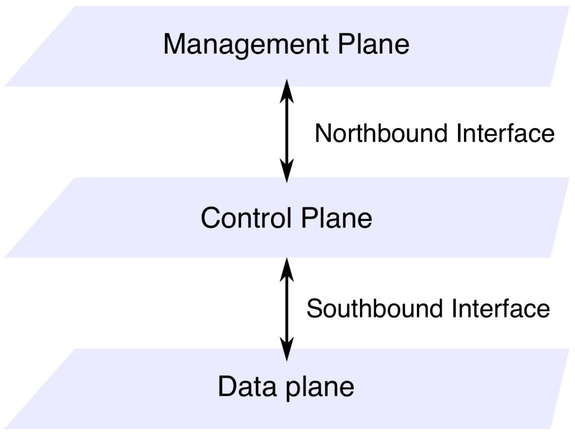


Figure 2.1: SDN consists of three different planes connected through two interfaces.

plane, and management plane interconnected with the southbound and northbound interfaces.

2.1.1 Data Plane

In a bottom up fashion, the data plane is the forwarding device interconnected through wired or wireless means. The data plane's purpose is to forward packets as quickly and efficiently as possible. One way traditional OpenFlow switches (i.e., the data plane) provide these forwarding properties is through Ternary Content-Addressable Memory (TCAM) hardware. Our enterprise research looks to push the forwarding functionality to end-hosts, which lack TCAM hardware, but because end-hosts must already maintain and manage state associated with network connections, our approach introduces negligible overhead.

OpenFlow controls forwarding devices by maintaining flow tables. The OpenFlow data plane maintains one or more of these flow tables. Each flow table is comprised of three different components: rules, action, and statistics. The rules are a set of matches on various network-related fields such as protocol ports, IP addresses, and VLAN tags. For each rule there is an associated action. The action tells the data plane how to forward the packet. The action could be be to drop, forward, or even modify the packet inline. Finally, for all rules in the flow table, statistics are kept such as the number of packets and bytes triggering a certain rule.

2.1.2 Southbound Interface

The southbound interface is how the control plane and data plane communicate and determines the communication protocol between the data plane and the control plane. It also defines the instruction set that the data plane will use to control forwarding of packets. In our work, the southbound interface protocol is OpenFlow. Other protocols such as Forwarding and Control Element Separation (ForCES) [86] exist, but they lack OpenFlow's level of support.

2.1.3 Control Plane

The SDN control plane, often referred to as the controller, is the component that programs and manages forwarding devices over the southbound interface. Accordingly, the control plane is required to understand the OpenFlow protocol. In addition to speaking the OpenFlow protocol on the southbound interface, the control plane also speaks over a northbound interface, which is discussed in the next section. The control place is comprised of a "network operating system" (e.g., Floodlight [12]) and all the built in components offered by the network operating system.

Many OpenFlow controllers exist in the research community and industry today. Industry often has a preference towards towards building their own SDN controllers [21], but many open source controllers also exist [3, 12, 28, 31, 89]. While these open source controllers support the OpenFlow protocol, each controller has particular characteristics that should be considered before deployment. In particular, the programming language the controller is written in is indicative of the performance. The POX controller is written in Python and is inherently limited due to the lack of true multithreading functionality in Python. Accordingly, its performance can be considered subpar with a processing rate of around 35,000 packets per second versus other controllers that process more than 100,000 [89]. However, there is a tradeoff between performance and usability with any controller. As result, our work leverages multiple controller types depending on the application at hand and whether flexibility in design or better performance is desired. However, any controller implementing the OpenFlow protocol can be used.

Some controllers may provide different built-in functionality. For example, Floodlight provides several built in modules such as a topology manager, device manager, and link discovery. Such functionality is extremely useful for certain applications. Rather than requiring developers to build a custom topology manger, the controller can determine the topology and provide a uniform API to all management applications. These controller-supplied modules typically run in the same privilege zone as the other management plane applications. As a result, applications provided by other developers can influence the state of built-in controller modules. This relationship has been the focus of recent security research showing that existing controllers are vulnerable to poisoning attacks [111] [157]. Controller security will become increasing relevant to our research as our work grows and begins to incorporate controller modules that may be developed by other entities, such as device manufacturers.

2.1.4 Northbound Interface

The southbound interface must be a well-defined protocol, such as OpenFlow, in order to allow the switch and controller to communicate, However, the northbound interface is provided by the controller and is an API offered to the management plane. This interface allows developers to write higher level applications to control the data plane. The API may also provide information on abstraction layers run by the control plane, such as topology discovery, as previously discussed. It is important to note that the northbound API is offered by the controller, and as such, this API may not be standardized; standardization remains an open question [83]. While this question is out of scope for our work, we note that it hinders a straight-forward conversion of management applications between controllers.

2.1.5 Management Plane

Applications that leverage the northbound interface to control the data plane are referred to as the management plane. The management plane runs more traditional applications such as firewalls, routing, and other applications that enforce policy. These applications may maintain their own state in addition to leveraging the control plane's API for information such as the current network topology.

Each management plane application registers with the controller what OpenFlow events the application wishes to consider. Some applications may act in a proactive manner in which rules can be preemptively pushed to the switch. For example, a layer 2 firewall may only be interested in new connections from switches in order to push static flows and will require no further interaction with the switch. Applications, such as a load balancer, need to act in a reactive manner where network packets are sent to the controller dynamically to be evaluated. Management applications need to introduce minimal delay when processing packets. A delay in processing new packets will cause queueing delays to increase. Accordingly, applications requiring longer term, in-depth analysis, such as an intrusion detection system (IDS) must insert flows that cause dedicated middlebox processing. Packets are processed sequentially in the management plane by each application. For added efficiency, if any management plane application determines unequivocally how the packet should be processed or is the only application requiring the packet, that application can push the corresponding OpenFlow rules and stop other modules from wasting cycles handling the packet.

We present a more detailed view of SDN consisting of a single controller and switch that uses the OpenFlow protocol in Figure 2.2.

2.1.6 SDN Considerations

SDN can simplify network management and security while allowing dynamic reconfiguration, but it does not come without its own considerations. We briefly highlight a few considerations that should be taken into account for SDN deployments.

Controller Placement

The Controller Placement Problem [106] in SDN is important to the performance of the network since the location of a controller will affect the network's ability to respond to network events. The metrics surrounding the controller placement problem will vary between scenarios and will depending on factors such as latency, the number of hosts, geographic layout and available resources. In Chapter 8 we focus on the controller placement problem when dealing with residential networks.

Scalability

OpenFlow is particularly concerned with scalability at the data plane and the control plane. OpenFlow data plane scalability research focuses on switch performance with respect to the number of rules and actions that can be stored in a switch's TCAM. Traditional (non-OpenFlow) switches

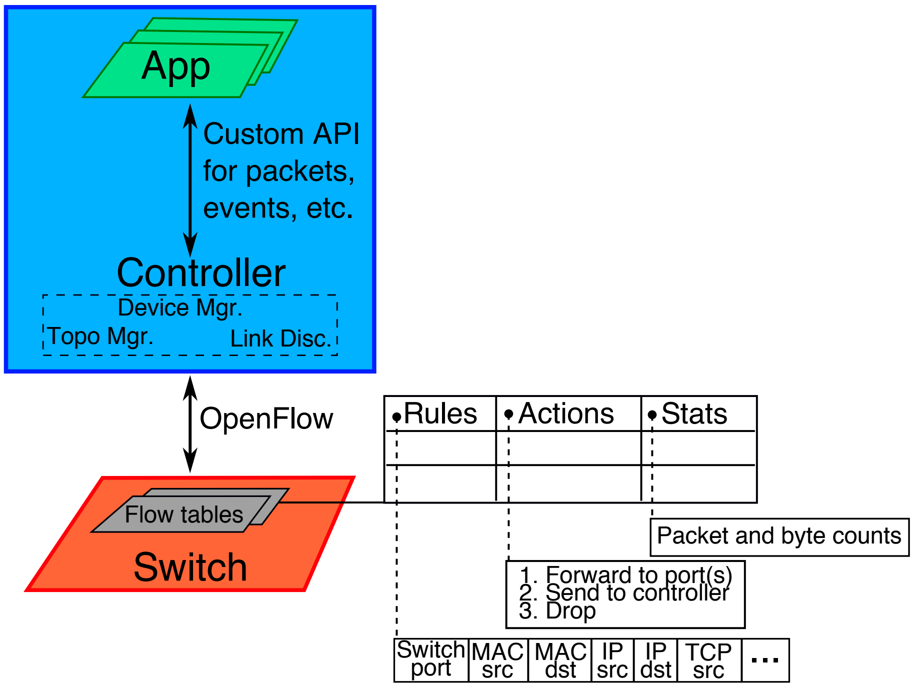


Figure 2.2: A more detailed overview of an SDN using the OpenFlow protocol.

typically only need to store course-grained layer 2 information such as media access control address (MAC) pairs. OpenFlow switches need to support rules, actions, and statistics. Rules alone can have over 12 different match fields [94]. With limited TCAM, OpenFlow switches can support between 750 and 2,000 flows, with an unspecified number of flows being stored in software tables [119]. Some enterprise grade OpenFlow switches have a maximum of 97,000 OpenFlow rules [98]. The inability to handle fine-grained flows scalably makes OpenFlow switches a target for denial of service attacks [153].

The control plane must also be scalable. If a logically centralized controller becomes unavailable, OpenFlow networks face the same challenges of other single point of failure systems. Namely, switches without a control plane are unable to forward packets associated with new network flows. The original OpenFlow specification did not support distributed controllers. Since OpenFlow's debut, other approaches have retroactively developed physically distributed but logically centralized controllers [84, 118, 170].

2.2 Network Outsourcing

The residential research in this dissertation is dependent upon the notion that network functionality can be practically outsourced. In a position paper, Feamster [91] first suggested outsourcing home network security to third-parties who are more knowledgable and better equipped to manage home networks. To outsource, the paper proposes using programmable switches. Programmable switches enable a third-party to aggregate data for all networks under its control. With aggregate data, distributed network monitoring and inference algorithms can then be used to, for example, detect distributed, coordinated behavior. Research challenges presented in Feamster's work include scalability, privacy, remote manageability, resiliency to attack, and policy conflict resolution. This dissertation work either directly attempts to address each of these issues or is indirectly addressed by way of leveraging existing work (e.g., SDN policy conflict resolution work). Gibb *et al.* [103] presented a more generalized outsourcing viewpoint that introduced the notion of a Feature Provider that would provide a Feature API to enterprises. Their prototype and evaluation included outsourcing a web cache and intrusion detection system.

Gibb's work lacked a thorough evaluation and left several unanswered questions. In particular, their work left unanswered questions related to performance and traffic redirection. A more thorough analysis of outsourcing was presented in [154]. The Appliance for Outsourcing Middleboxes (APLOMB) approach considers the question of whether it was possible to outsource an enterprise's network middleboxes to the cloud. APLOMB's motivations for outsourcing were to reduce infrastructure cost, simplify management, and leverage the cloud for dynamic scaling and failover mechanisms. In outsourcing, APLOMB focuses on meeting functional equivalence, low complexity at the enterprise, and minimal performance overhead. APLOMB finds that by using DNS techniques for cloud redirection and adding a specialized gateway at the enterprise they can meet their goals for nearly all enterprise middleboxes. Exceptions to this were internal firewalls that the gateway does not see.

Existing work has proposed outsourcing network functionality and the key challenges with doing so, and has presented preliminary work in the area. A commonality between past research research is that the enterprise is the core focus. In some instances, the techniques applied have no relevance in the residential network. For example, APLOMB considers multiple techniques for transparently inserting cloud-based middleboxes in path of enterprise network traversing the boarder gateway, but for the best performance, the chosen solution was to use a cloud DNS server to help direct clients through the middleboxes. In the residential environment, where most connections are outgoing, this approach is not feasible. Furthermore, performance considerations have been strictly limited to an enterprise, which has substantially different performance characteristics. This dissertation work explores similar questions with similar goals to the APLOMB architecture in the residential environment.

2.3 Network Function Virtualization

Network Function Virtualization (NFV) is an effort being pushed to move previously hardware dependent network applications into a virtual environment [25]. NFV would allow hardware specific functions such as, carrier grade network address translation (NAT), firewalls, and content distribution networks (CDNs), to be virtualized and placed in a general computing platform. Among several benefits, NFV aims to reduce operating costs and enable scalability.

NFV is frequently considered in deployment scenarios using SDN. SDN lends itself as a natural abstraction to chaining network functions together; however, NFV is a standalone movement from SDN and neither are reliant on each other. Our work, particularly in the residential network, intertwines the NFV and SDN abstractions.

2.4 Residential Networking

Feamster's [91] position paper first suggested outsourcing home network security to third-parties who are more knowledgable and better equipped to manage home networks. Project BISmark [163] is the most well-known body of work in residential networks and extends Feamster's position paper on outsourcing home network security to third-party experts. In comparison to our work, BISmark's deployment model is largely reminiscent of what exists today by hosting all necessary software on the home router itself. Further, BISmark has been used for extensive performance characteristics of residential network ISP's [164] and wireless performance [165] and has not thoroughly examined residential SDN. The only exception to this being a data cap enforcement application which only needs to probe a router regularly for information to handle enforcement [116] and did not consider the implications of controller placement outside of the LAN. Finally, BISmark performance results have not examined connectivity to public cloud infrastructure or the implications of hosting a cloud-based controller or NFV middleboxes.

Other research has also focused on deploying SDN within the home. The Homework project [134] uses a PC acting as a router running an OpenFlow controller within the LAN to understand HCI aspects of LAN network management. Yiakoumis *et al.* [178] proposed using the FlowVisor [155] tool to allow an ISP to facilitate the management of certain home devices by outside service providers, such as utility companies and later created an approach for users to express QoS preferences to the user's ISP [177] . Yiakoumis's work measured the impact of an OpenFlow controller that was within 16ms of 7 homes and does not provide broad analysis of data on viability for nation-wide deployments of SDN and NFV solutions. Similarly, Gharakheili *et al.* [102] allows a home network user the ability to express bandwidth restrictions for QoS/QoE with ISP participation, using an API with a cloud agent. This work is silent on both performance and support for other applications. Lee *et al.* [123] suggest using cloud-based SDNs for auto-configuration and identification of devices, but only considers the overhead of the configuration and identification protocol rather than all network traffic.

While some of these prior efforts have focused on outsourcing network management, they do not address significant deployability considerations or security concerns. In particular, they have not considered approaches to outsource security controls in an incrementally deployable way, nor approaches that allow users to be self-sufficient in doing so. In this dissertation, we instead focus on using SDNs to create an immediately deployable solution for specific applications without requiring support from entities such as the residential ISP. By sharing these applications and tools, we demonstrate that experts can create and share security tools with less technologically sophisticated users.

Chapter 3

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

3.1 Introduction

This Chapter begins by first rethinking how SDN is deployed in the enterprise. A host-based SDN deployment, as opposed to a switch-based deployment, enables us to address existing scalability concerns with SDN that may hinder real-world deployments. We also show how system-level context can be incorporated into network control information to improve the situational awareness operators have when attempting to understand network activity. By fully understanding the network activity from end-systems, network operators can mitigate security risks, such as data exfiltration of personal information including credit cards, the spread of malware or system compromises. In traditional systems, network operators are typically blind to intra-subnet traffic, since hosts directly forward the traffic without traversing security enforcement and monitoring devices. Recent innovations, such as the software-defined networking (SDN) paradigm, hold the potential to partially address the problem: with appropriately crafted fine-grained flows, the OpenFlow protocol [128], a widespread standard in the SDN community, allows a centralized controller to learn each time a new network flow is created.

While SDN approaches hold promise, they face two significant challenges: 1) fine-grained flows in OpenFlow's data plane controls do not scale to large networks [79] (described in Section 2.1.6) and 2) OpenFlow is inherently blind to end-host activities, since it operates in switches and routers.

Beyond scalability concerns, OpenFlow does not provide network operators with detailed visibility into the end-hosts operating on the network. The OpenFlow standard creates matches based on network headers, but this information may not be semantically meaningful. As an example, a popular video conferencing application uses ports 80 and 443 for communication [132], even though these ports are intended for HTTP or HTTPS traffic. Without application layer proxies or deep packet inspection tools, operators cannot determine the actual origin or destination of the traffic or correctly determine if the traffic is communication between a Web browser and Web server. Even more concerning, malware can take a similar approach to create connections that look like Web requests while actually communicating to exfiltrate information or for command and control [68]. Network operators need details about the host context surrounding the network request to make informed access control decisions.

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

To answer these questions, we embrace the "dumb network, smart hosts" stance. We take the OpenFlow agent functionality out of network switches and routers and instead place equivalent functionality in the end-hosts themselves, as shown in Figure 3.1. In doing so, we create an SDN approach that provides detailed host context and can scale to large networks while still yielding high performance.

Our contributions are the following:

- **Host context for all network flows:** We allow operators to craft detailed policies for flow authorization that include information about the applications creating the traffic. The approach is modular and allows the communication of arbitrary context. As an example, we created a policy that tracked applications and users and allowed only root-installed programs to access the network. We found that even this simple policy would successfully block multiple malware attack vectors while introducing low performance overheads.

- **Scalable, fine-grained flow-based access control:** We address the "southbound" or data-plane scalability concerns in OpenFlow by leveraging the distributed computing power of the end-hosts to apply the rules that hardware switches would otherwise be required to manage. This allows the hosts to apply fine-grain rules while allowing the hardware to apply coarse-grain rules, providing scalable, detailed network understanding. Even in our unoptimized setup, we found that hosts could create 25 new flows per second and established flows introduced no new constraints on the hosts, scaling far beyond the capacity of TCAMs in modern OpenFlow hardware. Further, the approach introduced only 38 ms of delay to flow establishment.

- **Integration of host context into OpenFlow:** We provide an initial look into how our approach of incorporating host context into network flow information can be adapted to work with the existing OpenFlow protocol.

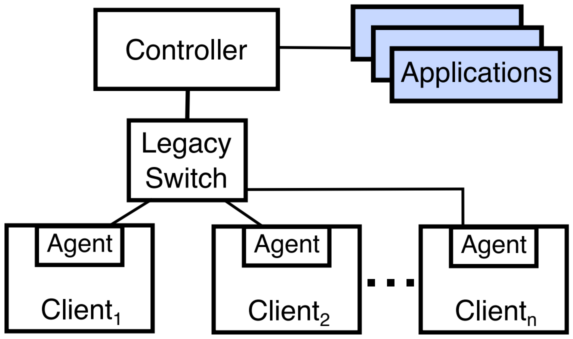


Figure 3.1: Integrating SDN functionality in host-based agents allows use of legacy switch infrastructure.

In creating this approach, we note that it can be deployed immediately and inexpensively using standard enterprise software deployment tools [133]. This allows organizations to incrementally adopt the approach with minimal effort and no capital costs.

3.2 Related Work

We briefly describe research surrounding host-context in SDNs for making informed decisions. Work related to SDN scalability is discussed in Section 2.1.6.

3.2.1 Extracting Context from End-Hosts

Ethane [72], an early SDN implementation, sought to enhance network security by allowing network operators to write detailed security policy that could include named entities such as users, end-host machines, and access points. Unfortunately, Ethane is a switch-based SDN approach, like OpenFlow, and it lacks information from the end-hosts that is needed to enforce the policy about users. Our work embraces the ideals of Ethane and augments it by instrumenting end-hosts and providing controls that allow policy enforcement using named entities, such as users and applications.

HoNe [93] provides process attribution by correlating network traffic to processes. The approach lacks centralized coordination and does not support arbitrary host context or embrace the SDN paradigm. Dixon *et al.* [85] use virtual machines and TPMs to allow network administrators to securely push network management to the end-hosts themselves but they lack situational awareness inherent to OpenFlow based SDNs. Parno *et al.* [141] present an approach called Assayer that uses end-host TPM capabilities to explore performance and security aspects of networks where the end-host verifies state already being maintained locally (e.g., number of packets sent) rather than requiring another device to determine the state manually. Participating systems push policies to off-path verifiers that supply clients with tokens to allow continual communication. Assayer's approach does not follow the OpenFlow SDN model, is reactive, and does not scale when attestation is required on a per packet basis.

Naous *et al.* [136] proposed a revision to the `ident` [115] protocol to allow a remote system to query for details about the application and other information associated with a flow. The authors designed `ident++` to work under the OpenFlow protocol to allow network operators to delegate administration of end-hosts from a centralized operator to local operators in the network. Our work shares the goal of fusing end-host information with network control with `ident++`. However, `ident++` does not describe or evaluate an implementation of the approach nor does it indicate how it would overcome the inherent scalability concerns of fine-grained flows in a switch-centric SDN architecture. In our approach, we take a host-based approach to address scalability. We then create and evaluate an implementation of the approach, both using a native OS solution and using a bump-in-the-wire implementation.

Other approaches have focused more on the context available on an end-host and how to extract this information for automated systems to better understand a user's workflow. These works can augment our approach and range from collecting mouse-clicks and keyboard presses [78] to application-specific implementations such as the user's interaction with a web browser [126,181]. Each of these approaches can be used to inform the host-based agents in our architecture, providing more context on the system's operation and enable stronger policies to be written.

3.3 Threat Model: User-Level Adversary

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

- **Trusted Operating System:** We allow regular user accounts to be compromised, but we assume a root-level compromise cannot happen, which would allow an adversary to change how the operating system kernel functions. Most host-based defenses, including anti-virus software, software firewalls, and host intrusion detection software assume that a system compromise only occurs at the regular user level, consistent with the best practice of "least user privilege" [151, 171]. We share this assumption, but note it may be relaxed using techniques such as trusted computing hardware or virtualization with trusted hypervisors. Even without such innovations, our approach can directly address many common user-level compromise attacks.

- **No Physical Attacks:** We focus on an adversary that lacks a physical presence inside the organization; otherwise, an adversary could sabotage systems and or use custom hardware to bypass our implementation. While we may address physical attackers in the future, we note that many attacks are launched remotely.

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

3.4 Approach: SDN via Host Agents

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

In describing our approach, we provide details of our reference implementation on the Ubuntu Linux operating system. While the details of the approach will vary across operating systems, the concepts are consistent and similar functionality may be available. To enable communication between agents and the controller, we used Python and asynchronous messaging provided by the Twisted framework [172]. These components were not optimized for performance and thus are conservative estimates of what would be possible in a production implementation.

While the approach is intuitive, it achieves powerful outcomes. Our system not only replicates the elevation and caching paradigm of traditional OpenFlow, it further supports features analogous to the "actions" in OpenFlow [44]. Some details of the behaviors may differ slightly as OpenFlow resides within network switches capable of controlling the datalink layer. Being a host-based implementation, our approach natively works at the network layer and above, though we do have the ability to influence datalink layer actions. We note that in addition to replicating the OpenFlow functionality, our approach scales even to extremely large networks as the end-hosts only store entries for their own flows and the logically centralized controller can be physically implemented using distributed controllers [170].

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

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

We now describe how we achieve each of these outcomes.

3.4.1 Host Agent: Intercepting Packets

Our host agent does not require special kernel or application modifications. However, the agent does run with administrator privileges, allowing it to manage the system's configuration and operation. Similar agent-based system administration tools are popular in large enterprises [133] and the agent software can be installed as a system service using traditional enterprise software deployment mechanisms. As a result, organizations can quickly and easily deploy the technology across parts or all of the enterprise network.

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

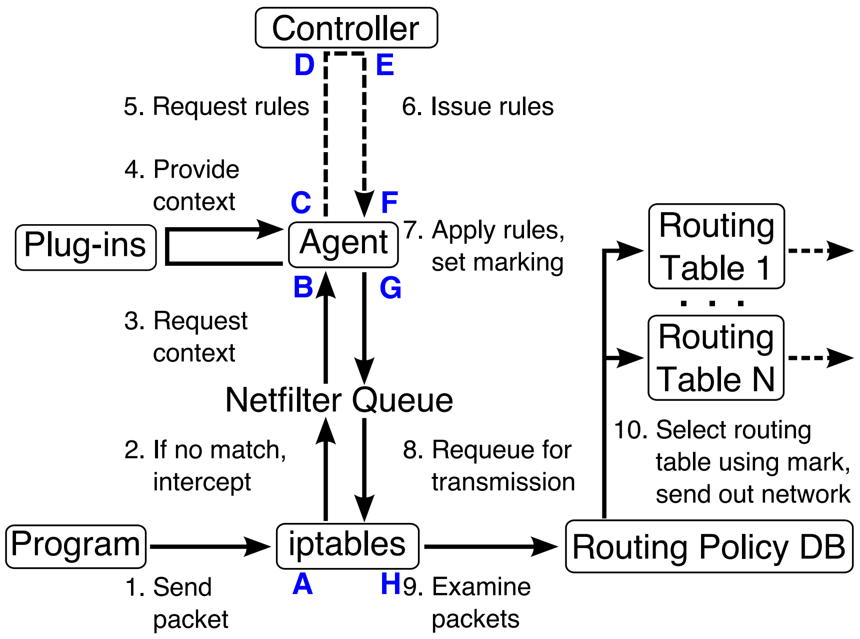


Figure 3.2: Overview of kernel and agent communication. Dashed lines represent network traffic while solid lines represent intra-system communication. The bold, blue letters indicate measurement points for the performance evaluation in Table 3.1.

In Figure 3.2, we provide an overview of how the SDN agent manages an end-host. When two communicating parties are using our approach, the process shown in Figure 3.2 is completed by both participants. In this process, the initial packet transmitted will not match any existing approved kernel flow, which is specified using the network layer addresses, transport protocol, and transport layer ports. Accordingly, while the packet is queued for transmission in the OS kernel, our SDN agent will extract the packet from the kernel queue.

Once the agent has intercepted the application's packet, it analyzes it and determines the context for the communication (Figure 3.2, steps 3 and 4). The agent is extensible and can encode and transmit arbitrary host context from any data source on the end-host. As an example, the agent may determine the owner and executable path associated with the process and provide this context. The agent then transmits a message to the SDN controller (Figure 3.2, step 5), which contains the flow tuple and the extracted host context, and requests instructions from the controller and, if desired, the packet payload as well.

Once the host agent receives a response from the SDN controller, the host agent will install appropriate NAT, firewall, routing and forwarding rules supplied by the controller (Figure 3.2, step 7). The agent then indicates the flow should not be diverted to the agent in the future. In our Linux implementation, we use a temporary `iptables` rule to update the marking for the flow to indicate the flow is authorized. The agent then signals `netfilter_queue` to release the packet, providing the altered version if requested by the controller.

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

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

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

3.4.2 Host Agent: Extracting Operating Context

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

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

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

3.4.3 SDN Controller

While our work focuses on modifying end-hosts to provide greater context to the network controller, the controller itself is an important consideration. In future work, we plan to modify the agents to speak the OpenFlow protocol, allowing us to use a standard, high-performance OpenFlow controller with applications that use the host context for decisions. However, our current implementation is a Python controller that interprets fairly simple policy and pushes rules to the host agents.

3.5 Security Enhancements: Contextual Policies

The increased visibility and control inherent in the fine-grained flows we enable can directly empower security systems [97,100,121]. Further, our approach enables new network security policies. We now describe such policies and their potential.

Network operators can use a variety of contextual languages, such as `POL-ETH` [72], Flow-based Security Language (FSL) [107], and Flow-based Management Language [108], to specify the high-level policies for a network. While these policies are amenable to formal analysis, their current instantiations are unable to distinguish among multiple users on a system. While prior work proposed such differentiation in the future [136], to our knowledge, our effort is the first to actually do so. Further, our approach provides additional contextual information from the end-host that was not considered in some of these prior efforts.

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

1. **Allow Administrative Processes:** If the process requesting network access is owned by user ID 0 through 999, grant access.

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

3. **Default Allow:** Allow network access by default.

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

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

3.6 Evaluation

To demonstrate and evaluate our approach, we create an implementation in a small network of virtual machines (VMs). These VMs run on a single server with 16 cores operating at 2.8 GHz and 64 GBytes running a KVM hypervisor. Each client system is allocated a single core and 512 MBytes of RAM. The network controller is allocated two cores and 2048 MBytes of RAM. All machines use Ubuntu 14.04 Server as the host operating system. For timing analysis, each host runs an NTP client and the VM server's host operating system runs an NTP server to keep the VM clocks synchronized. Each host has `iptables` preinstalled and we load the `conntrack` kernel module to allow fine-grained manipulation. The hosts are configured to ignore ICMP redirect messages, which can be generated when an intermediate hop is specified for a connection between hosts in the same subnet. Though enabled by default, ignoring such ICMP messages is a good security practice [65, 125].

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

3.6.1 Host Agent Performance

When considering an SDN system, the performance of the SDN agent (the data plane) and controller (the control plane) are the key considerations. While we perform basic performance measurements of our unoptimized SDN controller, our primary contribution is enhancing the data plane. Prior work that focuses on SDN controller scalability [170] can likewise be leveraged in our approach.

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

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

Table 3.1: Component-wise characterization of latency overheads over 1,000 TCP connections. Columns 2 and 3 correspond to the bold, blue letters in Figure 3.2.

	Fig. 3.2 Steps		Median	Std. Dev.
Component Description	Start	End	(ms)	(ms)
Initial Interception	A	B	0.088	0.105
Obtain Host Context	B	C	6.803	1.435
Elevation to Controller	C	F	3.535	1.688
Controller Decision	D	E	0.005	0.002
Marking	F	G	3.976	0.487
Re-queuing	G	H	0.022	0.005
Overall End-to-End	A	H	16.72	1.403

the latency overhead, we rapidly spawn new flows on the host agents and compare the results to those in traditional OpenFlow.

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

In Table 3.1, we show the latency introduced by each step of the process. We see that our SDN agent incurs a median of just under 17 milliseconds, with a significant portion of that time being devoted to gathering the host context. Further, this overhead is only incurred at the beginning of the network connection and thus may have little impact on actual applications since it is during the traditional connection build-up phase (in, for example, TCP's slow start).

To better understand the performance overheads, we performed high resolution timing on the hosts. We recorded the clock timestamp at each of the locations of the elevation process indicated by the bold, blue letters in Figure 3.2. We performed these timings on one of the hosts and the controller using `ovs-benchmark`'s [52] batch mode to create $1,000$ sequential connections. For each connection, the policy presented in Section 3.5 was enforced based on the context gathered on the end-host. To avoid introducing inaccuracies from nested timings, we conducted additional trials for the timings of the overall end-to-end timings with all intermediate timing samples disabled. We present the results of the timing experiment in Table 3.1.

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

Table 3.2: Round trip times with each host transmitting 1,000 packets.

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

3.6.2 Scalability of the Controller and Agents

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

We present the results of our scalability tests in Table 3.2. As expected, the median RTT numbers are roughly double the end-to-end results from Table 3.1 because both the initiator and the responder must contact the controller for approval of the flow. In the case of 28 hosts, the over-subscribing of the CPU cores did indeed introduce timing artifacts. When considering traditional OpenFlow using an Open vSwitch to connect two hosts, the flows per second are 243.3 and the median latency is roughly 4 milliseconds. While Open vSwitch has years of development and is built using a compiled language, thus achieving better performance, it is unable to provide the context we can provide in our approach. With further optimizations, our approach may yield more competitive performance.

In our scalability tests, we induced roughly 350 flows per second (14 hosts) with each host creating approximately 25 new flows per second. This new flow rate greatly exceeds the rate in Ethane [72], which induced less than three new flows per second in the worst case. Importantly, unlike OpenFlow or other hardware switch-based SDN implementations, all the data plane flow state is stored at the hosts themselves, eliminating any network constraints on the number of established flows. In essence, the number of flows created per second and the total number of flows a host may have are limited only by the computational resources on the host and the amount of time to vet the request at the controller. Our timing results show that our controller can handle around 200,000 new flows per second by spending around 5 μs on each packet. In practice, the connection processing overheads may decrease this value. The POX controller, which is also implemented in

Python, can only handle around 35,000 packets per second [89]. Accordingly, we do not expect the examination of host context in our approach will significantly degrade the controller's scalability.

3.6.3 Evaluating Policy Enhancements on Security

In Section 3.5, we provided an example policy for the network controller. In it, the controller will only allow regular users to create network connections if the process was created from a root-installed program (e.g., `/usr/bin/`). We now evaluate whether such a policy would be able to thwart persistent user-level malware.

We first perform an experiment using a simulated Linux malware called n00bRAT [54]. The executable provides an adversary with the ability to connect to a compromised machine and run preconfigured commands such as grabbing `/etc/passwd` and exfiltrating it. We modified the malware's source to run on a non-privileged port to match our threat model of user-level compromises. Accordingly, any commands preconfigured in the malware that require root access will be denied by the OS when attempting execution. The malware can be delivered through multiple vectors, including as an attachment in a phishing message or as a drive-by download on a vulnerable Web browser. In evaluating our policy, we test a case where a user on a host (that implements our approach) receives and runs the malware from an email attachment. We also perform a browser-based attack using Metasploit [53] and launch the malware using the compromised browser. In both cases, the malware is denied network access using our simple policy.

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

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

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

3.7 Discussion

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

3.7.1 Partial Deployment

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

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

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

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

3.7.2 Compatibility with Non-Linux Hosts

Our initial implementation uses the Linux kernel, but it can be applied natively on other operating systems, such as Apple's Mac OS X and Microsoft's Windows OS. Mac OS X's built-in firewall, `pf`, is based upon OpenBSD's firewall implementation by the same name [81]. BSD systems provide a special socket interface, called divert sockets, which can be used to intercept packets for the host agent. Such systems provide additional support for packet tagging rules and policy based routing, which are the remaining features needed for the host agent. In Microsoft Windows, the Windows

Filtering Platform [131] may provide the needed support for host agents, but further exploration is needed for conclusive results.

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

3.7.3 Potential for Network Security Policy

In a 2013 study of vulnerabilities on the Windows platform, researchers found that 96% of the critical Windows vulnerabilities and 100% of the Internet Explorer could be eliminated by removing administrator rights from the user's account [43]. Further, malware that injects itself into processes, such as the Zeus botnet [68], will be unable to inject into long-lived system processes as a user. Instead, the malware would only be able to inject less persistent user-level processes. While browsers are an attractive target, since they typically have regular authorized access to the network, other exploits, such as malicious code in PDF or word processor documents, may be more easily thwarted by policy since those applications rarely engage in network connectivity. Even in Web browsers, policies preventing certain traffic, such as SMTP communication, can constrain the abilities of injected malware. Simply having insight into the responsible application can greatly enhance organizational network policy.

Our system also enables policies for the graceful degradation of mission-critical systems faced with a user-level compromise. Organizations must make strategic choices about dealing with compromised hosts on their networks. From a security perspective, it may be appealing to immediately remediate any compromised systems and restore from backups. This can compete with the desire to preserve forensics for prosecution or for counter-intelligence [56]. In other cases, organizations may have practical constraints that hinder remediation efforts, such as running mission-critical services on the machine, essential on-going data collection, or even a simply constrained support staff for the organization. Unfortunately, in traditional networks, the choices can be rather limited: 1) isolate the host or 2) allow the host to communicate arbitrarily. The former approach may hinder mission continuity while the latter approach may introduce unacceptable risks for an organization.

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

3.8 Towards an OpenFlow-Compatible Host-based SDN

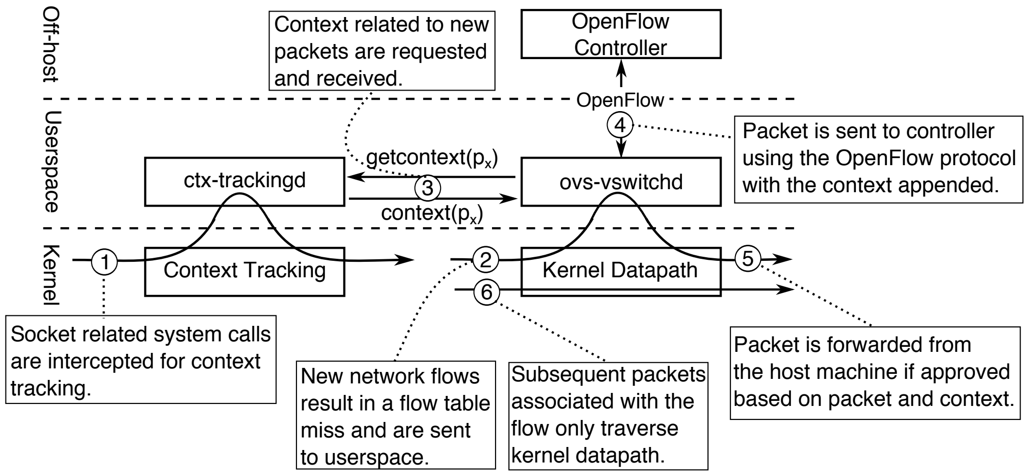
We have shown that host-based agents can be used to scalability integrate SDN functionality to hosts instead of switches. While showing the scalability of the approach, we also introduce the notion of incorporating host-context into the SDN functionality to provide network operators with better situational awareness of their network as well as the ability to create more powerful network policies than exists in other SDN frameworks. To achieve our goals, we proposed a "clean slate" solution where we implemented a new SDN protocol and implementation. Unfortunately, clean slate solutions can hinder large scale adoption [147]. Accordingly, we now discuss an initial look at how we could modify our approach to incorporate host context into an existing SDN protocol, OpenFlow, thats deployment scenario is not targeted towards end hosts and does not support host context.

3.8.1 Integrating OpenFlow into Host-based SDN

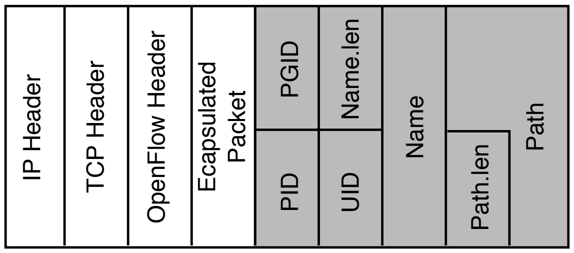
A requirement of our integration effort is to be backwards-compatible with existing OpenFlow infrastructure and remain incrementally deployable. Our integration should provide previously supported system information, such as the process ID and user ID, while also providing an extensible message format that can include GUI context information as discussed more in depth in Chapter 4.

We realize our goal of integrating into the OpenFlow protocol by first replacing our host-based agent with an OpenFlow-capable agent called Open vSwitch (OVS) [144]. OVS is an active open source project that is well established in the OpenFlow research community. By replacing our custom agent, we gain both the performance benefits and the support of the OVS development community. OVS is also OpenFlow compliant in versions 1.0-1.4 of the OpenFlow specification. Accordingly, integration efforts into OVS will fast-track progress as we will not need to develop OpenFlow support ourselves. Figure 3.8.1 shows the technical aspects of our integration efforts.

The changes necessary for Figure 3.8.1 are two fold. First, we create an active context tracking application called `ctx-trackingd`. An active context tracking approach is necessary since our existing implementation of reactive context gathering would significantly reduce OVS's performance. `ctx-trackingd` has a userspace and kernel component. Second, we modify Open vSwitch's `ofputil_encode_ofp10_packet_in()` function to ask `ctx-trackingd` for context related to a network packet going to the controller and then append the context to the end of the packet. Fortunately, this approach is transparent to existing OpenFlow management plane applications since the context always comes after the original encapsulated packet. This allows us to maintain backwards compatibility.


Off-host
Userspace
Kernel
Context related to new packets are requested and received.
OpenFlow Controller
OpenFlow
4
Packet is sent to controller using the OpenFlow protocol with the context appended.
getcontext(p_x)
ctx-trackingd
3
ovs-vswitchd
context(p_x)
1
Context Tracking
2
Kernel Datapath
5
6
Socket related system calls are intercepted for context tracking.
New network flows result in a flow table miss and are sent to userspace.
Subsequent packets associated with the flow only traverse kernel datapath.
Packet is forwarded from the host machine if approved based on packet and context.

(a) Host modifications



(b) Packet modifications

Figure 3.3: Figure (a) shows how we will perform host modifications to `ovs-vswitchd` to append context to OpenFlow packets. Figure (b) shows the context appended to the original OpenFlow packet where the shaded portions represent host context.

3.8.2 Performance Analysis of a Contextual Host-based

Using the same experimental methodology as described in Section 3.6.1, we show the performance results of our SDN approach using OpenFlow and leveraging OVS as our host-agent in Table 3.3. When compared to Table 3.2, we see that OVS's performance with proactive context gathering results in over 3 times the number of flows per second with host context enabled. When compared to a pure OVS deployment, incorporating host context only results in a 16% reduction in flows per flow.

3.9 Concluding Remarks

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

We created a prototype implementation and evaluated it in a real physical network with systems that lacked support for host-based SDN. We evaluated the approach at a higher scale using a virtual

Metric	Open vSwitch	Modified Open vSwitch
Median Elevation (ms)	1.98	2.739
Median RTT (ms)	6.25	7.390
New Flows/sec	103.19	86.80

Table 3.3: Table showing the performance characteristics of host-based SDN using Open vSwitch. Modifying Open vSwitch to support appending context results in approximately 16% reduction in flows per second. These results are incorporated from Najd *et al.* [135].

network. In doing so, we found that our approach incurred minimal overheads. We also briefly explore the possibility of integrating our approach into existing OpenFlow technologies. Leveraging Open vSwitch shows an improvement in performance over our clean slate approach.

Our work provides a foundation for potential future work. Future work includes exploring proxy solutions for legacy devices or assets not owned by the organization. Additionally, we will also examine how virtualization and trusted computing technology can be leveraged to relax some requirements in our trust model. Finally, we will explore building more advanced policies for enterprise network systems.

Chapter 4

Extending Host Context into the GUI

4.1 Introduction

In Chapter 3, we discuss our enterprise solution to securing networks using host-based SDN. By maintaining a presence on the end-host, we are able to better understand and attribute network interactions from a *systems* perspective. However, computing devices are designed to serve an end-user by allowing a user to complete tasks. To complete tasks, a user interacts with a machine using inputs to the graphical user interface (GUI). In this chapter, we seek to leverage not only system information when making policy decisions but also incorporate a user's interaction with an application's GUI to act as an endorsement for network activity.

Interactions with a GUI and the context generated from interactions determine how an application should behave. Accordingly, context and interactions may act as user endorsements for low-level operations such as network communication or disk access. With few exceptions, applications without an on-screen presence are not user-oriented. These applications may either be well-known daemons, such as clock-synchronization systems, or may be an indicator of unauthorized software that is attempting to evade detection.

For malware, it is advantageous to remain hidden in order to avoid detection. One way to remain undetected is to have no, or very little, on-screen presence. Indeed, the majority of malware never has an onscreen presence, and those that do, briefly present an error to avoid raising suspicion and lack subsequent interaction with the user [63]. As such, applications that have an on-screen presence should bring some legitimacy to an application and its behavior. User interactions with GUI objects, such as buttons, can either result in internal changes in the process's memory or could result in system specific operations such as opening a descriptor to a file on disk or creating a network connection. Importantly, these operations result from discrete steps taken by the user when navigating through the GUI interface.

Our goal is to leverage the inherent GUI structure, along with a user's input, to understand what underlying system interactions should occur as a user interacts with an application. Specifically, we aim to detect when applications perform network operations without a user's endorsement. We detect unexpected network connections by building *GUI signatures* of applications. The signatures are comprised of *paths* and paths are composed of GUI objects and the related interaction that must occur to transition to the next node in the path. For example, a mail client may have

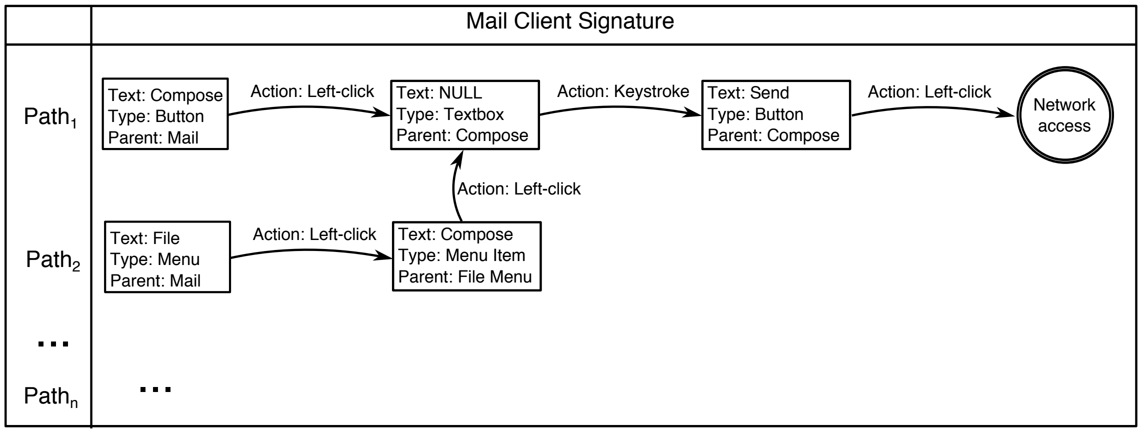


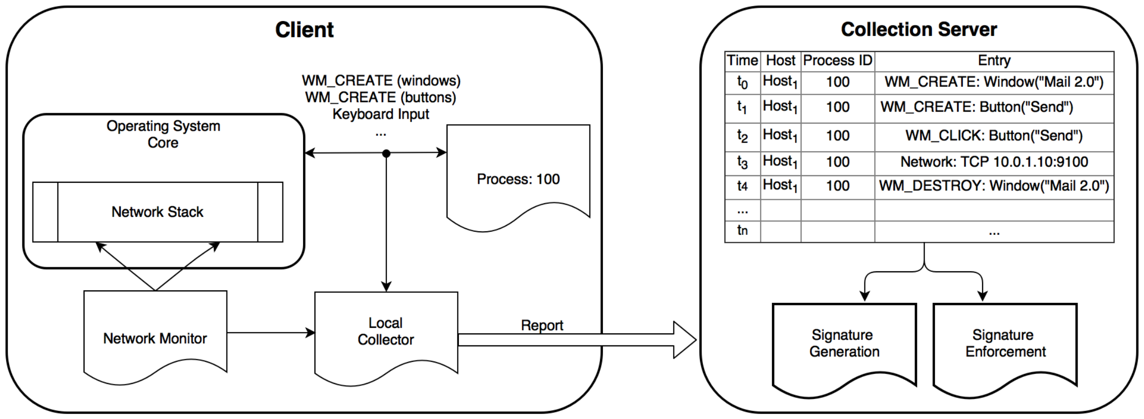
Figure 4.1: A GUI signature is composed of all valid interaction paths that result in network access.

multiple paths for sending an email. One path could be clicking on a "compose" button to create a new email, writing the email and providing the intended recipient's context information, and then clicking a "send" button. The collection of these paths creates a signature for a particular application as shown in Figure 4.1.

GUI signatures can be used to detect when applications attempt network access without an endorsement. Such unexpected access may be the result of something benign including asynchronous background traffic, such as application updates, which tend to happen infrequently and can be quickly ruled out as malicious. On the other hand, the traffic could potentially be the result of an attack resulting from a buffer overflow or backdoor in an application. Without understanding the user's interactions, the state of GUI, and the relationship between the two (i.e., the paths in a signature), it can be difficult to catch such attacks. It may be unclear to security applications, such as network-based intrusion detection systems, whether the network communication was solicited or not. With GUI signatures and an appropriate detection system, security operators can detect applications operating without the user's involvement or outside of a known path. GUI signatures also have the flexibility to allow operators to add more verbose signature paths based on information specific to an organization. Such extensions may include specific servers should be contacted when a user, for example, sends an email.

In this Chapter, we make the the following novel contributions:

- Introduction, motivation, and definition of GUI signatures and interaction paths.
- Discussion on how to build GUI signatures and an enforcement system on top of a modern operating system.
- A case study evaluation by detecting a ransomware application without a GUI and detecting a well-known text editor that has been compromised.



Time	Host	Process ID	Entry
t_0	Host_1	100	WM_CREATE: Window("Mail 2.0")
t_1	Host_1	100	WM_CREATE: Button("Send")
t_2	Host_1	100	WM_CLICK: Button("Send")
t_3	Host_1	100	Network: TCP 10.0.1.10:9100
t4	Host_1	100	WM_DESTROY: Window("Mail 2.0")
...			
tn			...

Figure 4.2: Our approach hooks into the Windows message passing system to determine what application objects exist and are being interacted with by the user and what network traffic the application generates in response. This information is sent to a local collector and stored at a collection server.

4.2 Approach: GUI Signatures

Our GUI signature approach targets the Windows platform due to both the number of deployments [29] and the number malware campaigns targeting Windows [20]. In Windows, applications with an on-screen presence communicate with the operating system via message passing as shown in Figure 4.2. Messages sent to and from an application dictate how the GUI should be created and what inputs are received. This level of abstraction simplifies the understanding of an application's objects being created, mouse input, and the translated version of raw keyboard input.

4.2.1 GUI Signature Components

GUI signatures can be used to detect when an application (1) is known but has no on-screen presence (i.e., a known empty signature), (2) is unknown and has an empty or non-empty signature, or (3) is known and has a valid GUI signature. We both build and enforce GUI signatures based on the information received by the collector. If the signature is known, we can determine if the application's system behavior, in our case network access, corresponds to some path in the signature. Signatures are composed of zero or more paths. Each path is composed the following:

- **GUI objects**: These are objects that ultimately create the visual representation of a user interface. This may include windows, menus, buttons, text boxes, or other types of objects. As a user interacts with an object, other objects may be created dynamically, such as creating a new window, and then dynamically destroyed. While they exist, a hierarchy of objects can be determined. For example, a text object may be the child of a button and the button the child of a window.

- **Interactions**: These are the interactions generated by the user. For an application to function, the user must interact with the GUI. Interactions can occur through the mouse or keyboard . As the user interacts with the application, the application transitions state, such as when new windows are created.

- **Sink**: The sink is the final node in a given path which represents the targeted system resource that is requested. In our work, the sink is always network interaction.

4.2.2 Signature Paths

As described, signatures are comprised of zero or more paths, which are themselves composed of GUI objects, interactions (transitions), and a sink node. We now describe how we collect each of these to form paths.

GUI Objects

A graphical user interface on a user's screen maintains a hierarchical structure defined by each individual application. Windows, menus, buttons, and text labels are some of the more common objects. When objects need to be created, an application uses the Windows API to request an object be created. It then receives a `WM_CREATE` message upon the object's instantiation. When destroyed, it receives a `WM_DESTROY` message. When created, objects have a known parent, such as the Desktop for newly launched applications, or may be children of previously created objects. As such, monitoring Windows messages allows us to understand the structure of a graphical interface. Windows also maintains *handles* to objects which allows us, given a handle, to access the object and immediately begin traversing up or down the hierarchy as well as accessing its internal state.

For understanding objects and their structure over time, we must be able to uniquely identify objects across executions. Although Windows maintains unique handles for objects, these handles are unique per instantiation. A handle will be different even within the same process execution if a window is destroyed and recreated. We create unique identifiers using the following naming convention: `object_ident = text:class:parent_text:parent_class:depth` where class is the class type of the object and depth is the object depth length from the root object of the process. In this naming convention, it is possible to have ambiguously labeled objects, but in our experience, such collusions occur infrequently. The ambiguity a collision could allow an attacker to take a non-network sink path that ultimately creates network traffic but would be undetected by our approach due to the ambiguity of a legitimate path also existing.

Interactions

GUI objects are created when an application launches and can be dynamically instantiated as the user interacts with the application. Understanding interactions is critical to determining when sink nodes (e.g., network traffic) are reached. In Windows, user input such as mouse and keyboard events are also relayed to applications using the message passing system. In particular, events such as the left click and keystrokes are passed to the application in conjunction with the handler associated with the object receiving the input. This information is then passed as a message. For example, a click on the `Print` button would result in a Windows message specifying the event type of `BM_CLICK` and a handle to the Print button itself.

By monitoring user generated events and the objects receiving the events, we can understand how a user interacts with an application. During the signature generation phase, we monitor all potential interaction paths that lead to a sink node. We later prune paths that do no result in reaching a sink node in order to reduce the number of total paths in the signature.

Sink Nodes

The final component in a path contained in a signature is a sink node. The focus of our work is to determine when an application should or should not be requesting network access based on a user's specific interactions. For this, we must link an application's network traffic and user interactions. Figure 4.2 shows that a local collector application monitors in realtime GUI events as well as monitoring network access in the form of new flow generation. Our approach focuses on detecting new network flows not initiated in response to the user's interaction rather than monitoring existing flow activity. As such, we avoid the added overhead and complexity of per packet analysis.

The collector records and links applications by their process ID. It records data in a time-series. As such, we know that any time the collector sees an application generating a new network flow that a sink node has been reached. Applications may not necessarily generate a single network connection per interaction path. For instance, a user clicking to send an email may result in multiple network connections such as a DNS request for the mail server and then the TCP connection for transmitting the email to the outgoing mail server. To account for these scenarios, we group the cluster of network connections as a single sink node using time-based heuristics.

4.3 Threat Model

Our GUI signature approach follows the same threat model established in Section 3.3.

4.4 Signature Generation and Enforcement

We now describe generating GUI signatures for applications and then how we use these signatures to enforce per-application network behavior.

4.4.1 Empty Signatures

Applications can be divided into two main categories as shown in Figure 4.3. Applications either come with a front-end GUI or not. Applications without a GUI cannot generate a signature for our approach. In these cases, we allow for a known, empty signature can be created. Since these applications do not interact with the user, our approach is unable to determine legitimacy of new connections. Therefore, we approve all connections from processes with a known empty signature (i.e., no GUI). We identify these applications with empty signatures by their full path.

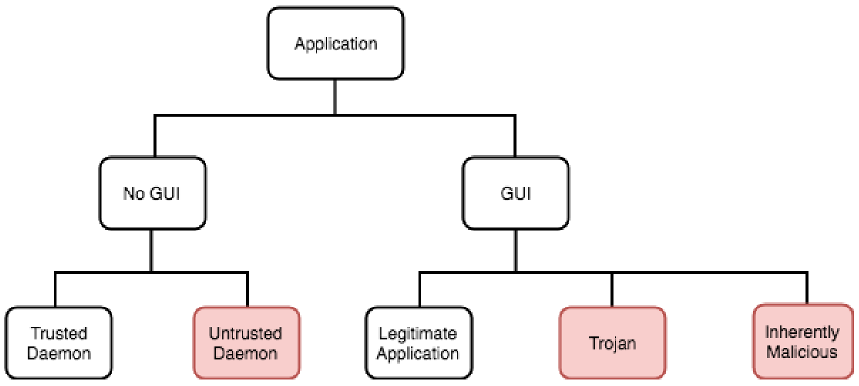


Figure 4.3: Applications can be broken down by whether a GUI component exists and further broken down into what is considered malicious or not. Red shaded boxes are considered malicious.

4.4.2 Generating Application Signatures

Next, we discuss building our signature database using applications with a GUI. In total, we consider 3 applications for signatures. More detail is provided on application signatures in Section 4.6. Building a database of GUI application signatures requires loading applications and generating potential paths. While in practice signatures could be built over time with passive data collection or automated GUI techniques [139], we chose to rapidly establish signatures by manually launching applications and exhaustively exercising all GUI possibilities. As a result, we will also generate all possible paths for a given application.

Our approach has the local collector running on an end-host report all monitored Windows messages and generated network traffic to a centralized collection server that aggregates data from all clients. The data collection can either be used for the signature generation phase or later used for enforcing derived signatures (Figure 4.2).

Sink Terminating Paths

Some paths reported by the collector do not result in a network sink. For example, opening the "About" dialog in an application may generate a path that does not terminate with a sink node if no network connection is generated. Depending on the application, the number of non-sink terminating paths will vary and will be excluded from the signature.

We determine paths that end in network sinks by monitoring user interactions with objects. Any time a user interacts with a GUI object, we report and collect the full path starting from object interacted as the leaf node through root node. Essentially, we start from the leaf object and traverse the hierarchy of objects until we hit the Desktop, which is the parent of GUI applications. This method allows us to determine the path required to reach an object. A common example is applications that have a print dialog, which can typically be reached through multiple interaction paths such as using the menu or hotkeys (e.g., `Ctrl+p`). Important to the printing process is that a print dialog window appears and the user left clicks (or uses the keyboard) to select the print button.

Figure 4.4 shows that a portion of the path when printing, for example, must be inferred.

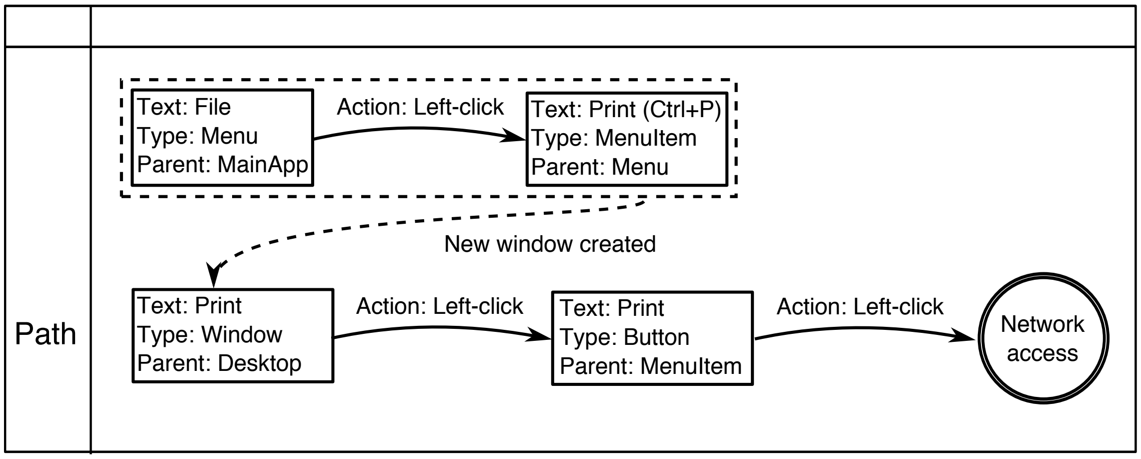


Figure 4.4: Common interaction path for printing. The dashed arrow and box represent the part of the path that is inferred.

Time	Host	Process ID	Entry
t_0	10.0.1.99	5561	Menu clicked: [(&File:Menu:Mail:#32770:2) → (&Mail:#32770:Desktop::1) → (Desktop::0)]
t_1	10.0.1.99	5561	Menu item clicked: [(&Print:MenuItem:File:Menu:3) → (&File:Menu:Mail:#32770:2) → (&Mail:#32770:Desktop::1) → (Desktop::0)]
t_2	10.0.1.99	5561	Window Disabled: [(&Mail:#32770:Desktop::1) → (Desktop::0)]
t_3	10.0.1.99	5561	Button clicked: [(&Print:Button:Print:#32770:2) → (&Print:#32770:Desktop::1) → (Desktop::0)]
t_4	10.0.1.99	5561	Network: TCP(10.0.1.10:9100:SYN)

Table 4.1: Data collected at the Collection server when a user clicks the print button on the print dialog screen for an application. GUI object entries are named using the structure of `object_ident = text:class:parent_text:parent_class:depth`. Double colons represent NULL values.

When applications create a new window as a result of interactions, the new window is frequently *not* considered a child of the parent application. Instead, the new window is a direct child of the Desktop. In many cases, we are able to determine inferred paths using a couple of different features. First, we know the new window is related to the original window interacted with because the new window does belong to the same process ID. Second, applications often disable the original window using the `WM_DISABLE` message. Thus, we can infer the previous interaction was a prerequisite for next interaction.

Figure 4.4 shows how some paths need to be inferred rather than directly obtainable by the hierarchical structure. In particular, some interactions create new relationships rather than following a direct parent-child relation. As with the print example, using the menu to print a document causes a new window to be created that is a child of the Desktop rather then the menu itself since the drop down menu is destroyed (visually removed) after choosing print and thus cannot have child nodes. Table 4.2 shows how we are able to use data collected to determine paths that result in network activity. In particular, we see that by clicking the print button in the print dialog box a new TCP connection is attempted. By exhaustively exploring all possibilities in the GUI, we find all paths that result in network activity.

4.5 Implementation

Our approach consists of two major components. One module resides on the client and the other component runs at the collection server. Although we focus on the Windows platform, the network and GUI components of the client module is the only OS-specific portion of our implementation and can be tailored for each individual OS.

4.5.1 Client Module

The client module has two important subcomponents. First, we have a network monitor that is implemented as a Windows driver. The driver uses the Windows Application Layer Enforcement (ALE) portion of the Windows Filtering Platform (WFP). Our driver allows us to monitor the creation of all socket operations including TCP and UDP connections. We use ALE as opposed to non-ALE filtering approaches to allow us to monitor at a per-connection or per-socket level.

We also implement a dynamically linked library (DLL) to intercept all Windows messages for applications. Windows has over 1,000 different types of messages that can be sent to an application. Windows will send all of these messages to an application regardless of the application's interest in the message. However, if the application is interested, it will have written a function handler to deal with the message. Some events, such as the `WM_MOUSEMOVE`, happen frequently as a user interacts with an application but do not provide much inherent value. For example, simply dragging the mouse across an application can result in hundreds of movement messages but are not necessary intended as explicit input such as a mouse click may be. In total, we limit the number of messages actively monitored to 15. These 15 messages include application-driven and user driven messages.

A local collector can also be configured to receive input, over a local socket, from both the driver and DLL and report that information over a TCP socket to a centralized collection server.

4.5.2 Collection Server

The collection server is an application that receives information collected at each end-host and is reported over the network. The collection server stores received data in a database format.

Signature Enforcement

Signature enforcement happens on a per application basis any time an application triggers network traffic. The enforcement module traverses event paths reported to the collector in reverse chronological order. Users can perform interactions out-of-order. For example, a user could choose to search for an old email before choosing to send a recently composed one. However, the entire path includes the interaction of composing the email in addition to clicking the "send" button. This

Application	Application Type	# Potential Paths	# Paths in Signatures	Application Complexity
`Notepad` v1607	Word processor	24	5	Low
`Notepad++` v7.3.3	Word processor	336	24	Medium
`CryptoLocker`[1]	Ransomware	0	0	-

Table 4.2: Applications we use for signature generation and enforcement evaluation.

means we cannot simply look at the most recent interaction for enforcement. Instead, we must look through the complete interaction log to determine if a complete path exists even if reported in non-sequential segments.

We recognize the task of processing an interaction history at the collection server may become more complex as we consider a variety of applications. However, we do not believe this to be an intractable task. In particular, there are other Windows messages that we currently do not consider that may help piece together how a user is interacting with a GUI. `WM_DISABLE`, which we take into account, along with instrumenting `WM_SETFOCUS`, `WM_ACTIVATEAPP`, and others may help to more precisely track a user's progressions through a GUI.

4.6 Overview of Signatures

We explore several applications for building application signatures on the Windows platform. First, we consider `Notepad`, an application that is part of the Windows OS. `Notepad++` is an open source text editor that dramatically improves functionality provided by `Notepad`. We are motivated in choosing `Notepad++` as an application given the source code is available and modifiable for our evaluation. It is also an interesting example given the recent disclosure that it was the target of a government exploitation toolkit [27]. Finally, we include a malware sample that has no GUI signature but attempts network access to a Command and Control (C&C) server.

4.6.1 Methodology for Building Signatures

For both GUI applications in Table 4.2, we generate all GUI paths by manually interacting with the GUI. We then inspect the paths collected for those interactions that result in network activity. To generate all paths, we attempt to enumerate all possibilities in the GUI including interacting with menus, toolbars, and other windows. In the future, we plan to integrated automated GUI testing tools to handle this step automatically [139]. Where necessary, we provide input that will facilitate network traffic. For example, opening a file in `Notepad++` can either happen locally via disk access or over the network by providing a network address for retrieving the file.

After enumerating the GUI possibilities, we parse the data collected at the collection server, which is stored as a time-series. We then use entries of network access as a starting point for determining the path associated with triggering the network traffic. Since network traffic is associated with a particular PID, we use the PID to filter which application GUI events and interactions. We follow create, destroy, and disable events to determine the start and end of a particular path and

[1]SHA256: 62f199dedfffef4eb71c33bdf22f4a9b3276f8a831999788059163fae43db48e

then determine the user generated input to cause the specific transitions from the start to sink of the path.

Paths and sinks may not be in a one-to-one ratio. More concretely, an application may create more than one network connection as a result of a user's interaction. As a result, we use the temporal nature of the connections generated to group the number of connections into the sink node. A typical example of this is when sending an email that results in two network connections (DNS request and TCP connection). In this case, we group these two connections into a single sink. If for some reason the application were to generate three connections, our enforcement module should raise a error.

4.7 System Evaluation

Having built the GUI signatures overviewed in Table 4.2, we seek to evaluate the effectiveness of our approach with signature enforcement. We do this by first ensuring that our system can properly match valid paths in our signatures. When then explore (1) an invalid empty signature (non-GUI malware) and (2) running a compromised application that has a known signature.

4.7.1 Valid Signature Enforcement

After building our signature database presented in Section 4.6, we first verify our system's ability to enforce known signatures. In particular, we evaluated our system over both active and inactive periods of use. In addition to the signatures presented in Table 4.2, a collection period of approximately 24 hours found a total of 11 background processes that were classified as known empty signatures. We added these empty signatures to our signature database, and as a result, no false positives were found for known empty signatures. In the set of known empty signatures, we found the expected Windows daemons and other dedicated update components to GUI applications, such as Google's update client for the Chrome browser.

During evaluation of known GUI signatures, we validated our enforcement capabilities with a subset of the available paths for both `Notepad` and `Notepad++`. Specifically, we choose to attempt to open and save files to the network. Both of these interactions may either be triggered by navigating the GUI using the `File` menu or by by using the hot-keys `Ctrl+o` or `Ctrl+s`. Network activity for both applications is triggered after entering a network address to open or save the location of the dialog generated and selecting the `Save` or `Open` button. During the enforcement phase, we verify when there is or is not a valid path and defer further details since it is ancillary.

4.7.2 Example: Invalid Empty Signatures

Our first goal in evaluating unknown signatures is to determine if we can detect an application that attempts to evade detection by running in the background without a graphical interface. To do so, we infect a client with the ransomware `CyptoLocker`. In addition to lacking a GUI, `CyptoLocker` runs in the background and attempts to connect to a C&C server by first performing a DNS request and then attempting a TCP connection to receive commands. Upon seeing the TCP connection, the enforcement module sees that the `CyptoLocker` binary does not have a GUI signature and is not whitelisted in the known empty signature database. As a result, our enforcement module is able to immediately detect that the network connection is illegitimate.

4.7.3 Example: Backdoor with Known Signature

With our approach, detecting non-user initiated connections for applications without a GUI is relatively straightforward. It is more difficult to detect when applications with a GUI are misbehaving. We evaluate our approach's ability to detect invalid paths in known GUI signatures by modifying `Notepad++` to have additional, hidden functionality. In practice, this type of functionality could be the result of the application acting as a trojan horse or potentially by having part of the application hijacked as recently discovered to be possible in `Notepad++` [27].

We replace our original `Notepad++` binary with a version that instantiates network traffic using a path not existent in the signature. In particular, if the user follows the menu path "?" $\rightarrow$ "About" $\rightarrow$ "OK", `Notepad++` attempts to establish a TCP connection to server controlled by the researchers. While this simple modification is sufficient to trigger our enforcement module to detect an invalid path, an actual attack is likely to be more nefarious in nature and could attempt any number of malicious activities such as exporting a user's clipboard or other data exfiltration.

Indeed, our enforcement module begins checking for valid paths after detecting that `Notepad++` has triggered a network connection. Fortunately, the path leading to the network interaction is not one of the 24 paths in Table 4.2 and results in the enforcement module accurately detecting the invalid path.

4.8 Discussion and Limitations

There are a vast number of applications with varying capabilities and interface layouts. As a result, evaluating GUI signatures both qualitatively and quantitively is difficult. Some applications may have non-trivial connections between different GUI components, potentially non-deterministic functionality, use complex protocols such as web browsers using HTTP. These applications are inherently difficult to link together. In the future, we plan to look more closely at these examples.

In our evaluation, we considered two well-known applications, `Notepad` and `Notepad++`, that from a security perspective seem relatively simple and potentially straight forward to secure. However, each application's ability to reach out over network for reasons such as saving files means it is not always obvious whether the application should have access to the network. If, for example, we had embedded ransomware functionality into `Notepad++` instead of our benign network request, it would impossible determine if `Notepad++` should be accessing the network without understanding the GUI. Indeed, ransomware generally attempts to reach out and encrypt network file shares [32].

Attempting to open files over the network when providing a network address is a particular instance of a challenging problem our work faces. Specifically, we have the challenge that network activity may be dependent on specific input from the user. In `Notepad`, if the user enters a network path, network traffic should occur, but if the user chooses a local file, network access should not happen. One way of addressing this challenge is by extending the interaction component of a GUI

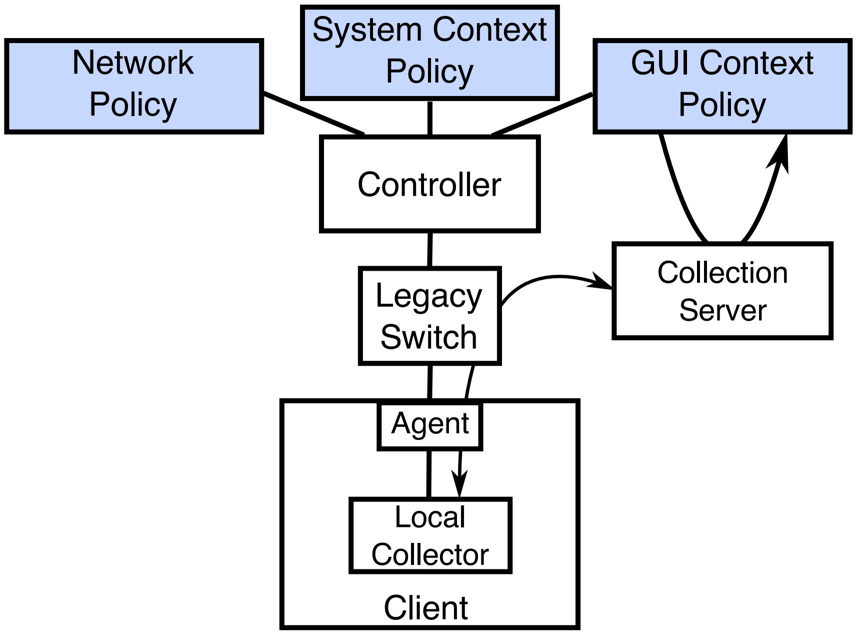


Figure 4.5: GUI context integration with host-based SDN.

path to include some input validation. In this work, we follow a "fail-open" scenario where ignore specific input and allow network traffic. We look to future work for better addressing this challenge. As a result, a sophisticated attacker could cause network activity on these multi-scenario paths and could avoid detection. Furthermore, an attacker could directly replace application-specific network activity with their own in attempt to blend in. In this case, the user would either experience a lack of functionality with the application or more network connections than expected would be generated, triggering an alert.

4.9 Integration into SDN

The Chapter has shown how we can use interactions with graphical user interfaces to act as endorsements for network activity. In the future, we plan to integrate GUI context into the decision making process of our SDN controller discussed in Chapter 3. In Figure 4.5, we show the GUI enforcement process could be integrated into our host-based SDN solution. Using our existing SDN approach, we can incorporate a new management application that queries the collection server based new connections are that generated.

In Figure 4.5, the GUI context policy application asks the collection server whether or not a particular connection should be allowed. The policy provides the collection server with necessary information including the host, PID, and network information. The enforcement thread then takes the provided information and searches for valid paths. The result of the enforcement thread and other policy applications then decides the fate of the network connection.

4.10 Graphical User Interface Related Work

There are three categories of research that are related to the efforts presented in this Chapter. The first is related to the idea that user interactions determine when network access should occur. BINDER [77] positioned that malware could be detected given that malicious software typically runs in the background and does not interact with the user. BINDER suggested a temporal approach to detecting uninvoked network traffic but failed to incorporate or link interactions with specific applications. Unfortunately, BINDER's approach is vulnerable to mimcry attacks [173] where malware can simply generate traffic at the right time in order to blend in. Similar to BINDER, Kwon *et al.* [120] attempted to detect botnet applications by attributing network traffic to user input, and while more detailed on interactions from users than BINDER, is still vulnerable to mimcry attacks due to an insufficient understanding of an application's interface. In contrast to BINDER and Kwon's work, we prevent mimcry attacks by linking specific interactions with particular objects of an application's GUI to specific network sinks that result from these interactions.

Bhukya*et al.* [67] suggest a technique to detect users masquerading as other users. Their approach gathers aggregate usage information such as mouse clicks and keystrokes. They then build per user profiles using machine learning and attempt to detect anomalies in behavior. While they are focused on a separate problem, both of our approaches recognize the utility in leveraging GUI information.

Other approaches have strictly looked at the network level understand inter-packet dependencies [176] and have also applied machine learning approaches to understand user-driven network traffic in browsers [182]. The results of network-based solutions to understand the cause and relationship of network traffic are promising but are inherently limited in that they rely on not only being able to understand the protocol itself but also that the protocol is unencrypted.

Several approaches have focused on access control with file management at end-hosts. Polaris [162] and [149] take a course-grained approach that involves replacing standard interfaces with custom interface components. By controlling the GUI, these works are able to precisely determine the underlying file operations. Shirley *et al.* [158] attempts to determine when applications are reading or writing to files that have no relationship to an application attempting to access the file. Shirley's approach requires maintaining a complete history, per application, of any file an application creates on installation or is created during the lifetime of the application. These approaches take advantage of user interactions with the GUI to understand the intent of the user with respect to file and access control. Unfortunately, they either require replacing existing functionality or indefinitely maintain history about an application. We are able to gather intent by hooking into existing API calls.

Finally, Gyrus [114] is a VM-based approach that tracks GUI input and ensures the input is not modified before being transmitted on the network. Gyrus provides secure text overlays (ran by the hypervisor) on top of untrusted applications such as web browsers. When the network traffic related to the text is sent, Gyrus checks to make sure the payload of the packet matches what was entered in the secure overlay. Gyrus is limited in: 1) its deployment model requires a VM to host the secure overlays, 2) the packet payload is unencrypted, and the most challenging 3) being able to understand how the original text is packaged by an application into a network packet payload.

4.11 Conclusion

In this Chapter, we look to understand how the GUI can be used to make security decisions about the legitimacy of network interactions. We find that it is possible to help make these decisions by understanding an application's interface and how the user interacts with that interface. We introduce the notion of GUI signatures, which are comprised of paths. Paths relate a GUI's objects and interactions in a deterministic way such that if the path is completed, network traffic is considered to be endorsed by the user. We show that our approach can be deployed on end-hosts and be used to detect malicious applications that lack a GUI and are not trusted daemons. Next, we show that GUI signatures are an effective approach to detecting a more complex and subtle class of attacks where legitimate applications have been compromised and exhibit network activity on unapproved paths. Finally, we discuss how this work fits in with the larger vision of host-based SDN by providing a new source of information for a centralized controller to consider when making real-time decisions about network traffic.

Chapter 5

Enhancing Residential Networks with Cloud Middleboxes

5.1 Introduction

In the enterprise setting, organizations often invest in innovative networking infrastructure and middleboxes to improve their network performance and security. These organizations may use security tools, such as hardware firewalls, proxy servers, and intrusion detection systems to fortify their networks. These networks are often administered by a dedicated IT staff with expertise in networking and security.

By contrast, in the residential setting, the networks are typically created by end-users who often lack expertise in computing, let alone in networking or security. Residential users often initially configure their networks and then neglect the infrastructure until it fails. This approach can yield networks that have relatively weak security measures and may yield sub-optimal network performance. This concern may grow in importance as more Internet of Things (IoT) devices, with varying security assumptions and weaknesses, begin using the residential network [46].

While it may seem appealing to simply introduce enterprise tools and techniques into a residential network, such an approach is impractical for multiple reasons. First, enterprise equipment and software is often expensive and would be unaffordable for many residential users. Second, enterprise solutions are designed for networking and security experts who actively maintain the systems; many end-users are likely to lack the technical proficiency or time to configure and maintain these tools. Finally, the needs for enterprises and residential users diverge in many practical ways. As an example, enterprises may focus on high performance and redundancy while residential users may focus on more mundane constraints, such as preferring computing equipment with smaller form-factors [163].

The goal in our work is to enhance residential networks with all the innovations of more sophisticated enterprise networks and enable residential specific innovations while minimizing complexity for end-users. In pursuing this goal, we focus on consumer-grade network router hardware that typically operates as the core of residential networks. These devices manage Internet traffic and traffic within the residential LAN, making them an ideal target for enhancement. Unfortunately, commodity routers are limited both in hardware and software capabilities. While the hardware is capable of serving a residential network, it is unable to support the processing power to perform tasks such as intrusion detection. As such, middlebox software solutions may not exist for home routers. Given these inherent limitations, it is intractable to integrate enterprise solutions into these devices.

We propose using software-defined networking (SDN), network function virtualization (NFV) and middleboxes in public cloud-based systems to provide enterprise services in residential networks. Our approach, called Residential SDN (ReSDN), will enable new, residential focused solutions. While prior work has suggested that such an outsourcing approach could address the limitations of consumer-grade routers [91], [179], these approaches typically rely on outside cooperation, such as the user's Internet Service Provider (ISP). Instead, we propose a solution where a user may deploy the approach without requiring ISP cooperation or ISP equipment deployment. To do so, we modify consumer-grade routers to use the OpenFlow SDN protocol and a custom agent to outsource management and control to a cloud-based controller. This allows the cloud system to provide a suite of enterprise services without the need to further modify the user's equipment.

After building our modified residential router, we develop two proof of concept security applications. The first application we build is a DNS blacklist that seeks to provide benefits similar to Google's Safe Browsing [49]. However, unlike Safe Browsing, our approach affects all devices at the residence, eliminating the need for host modifications or DNS infrastructure changes.

The second security application provides end-users with improved end-to-end security by reducing the number of paths taken across the Internet that are unencrypted. Movements such as HTTPS Everything [48] and Let's Encrypt [51] aim to increase the amount of Internet traffic that is being encrypted with SSL/TLS. However, some statistics find that over 60% of some network traffic is still unencrypted HTTP [50]. Our security application will use our foothold in the home to tunnel encrypted network traffic such as HTTP from the router to a cloud waypoint. The waypoint routes the unencrypted traffic to the destination. By strategically choosing the public cloud waypoint, we can reduce the number of potentially malicious hops an unencrypted packet takes to and from the destination.

We summarize our contributions as the following:

- **Immediately and Incrementally Deployable Residential Services:** We present our ReSDN infrastructure in which a user can independently employ the approach to achieve greater security without relying on ISPs that may not have incentives to cooperate.

- **Discussion of Two Real World Applications:** We discuss two security applications that our infrastructure allows us to deploy. First, we focus on building a lightweight OpenFlow controller module that acts as a DNS blacklist. Second, we discuss an approach that enables us to better secure end-to-end communication with servers that do not provide appropriate mechanisms (e.g., HTTPS) without client or server modifications.

- **Prototype Implementation and Performance Evaluation:** We have created a prototype implementation of the approach and have evaluated it in a residential setting. We implemented an example controller application in the cloud, a DNS blacklist, and cloud waypoints to enable secure communication between clients and servers even if the servers do not do so otherwise. We then present a security and performance evaluation of our work. We found that even our unoptimized prototype is viable for immediate consumer use.

5.2 Approach: Inspection in Cloud-based Middleboxes

To obtain all the advantages of enterprise middleboxes and NFV, we are modifying home consumer-grade routers to support OpenFlow and communicate with a controller in the cloud. In doing so, the controller gains full control over the network flows and can divert these flows through a series of virtual middleboxes, which can implement the functions of enterprise equipment in software. Since residential users have lower performance requirements, these software approaches can meet the user's demands while having flexibility in their use.

We now describe the key elements of our ReSDN infrastructure, including the modifications to the consumer-grade router including a software agent and the control systems in the cloud.

5.2.1 Consumer Router Modifications

To achieve our goals, we must modify the consumer's router to support the OpenFlow protocol, and unlike prior work, we must place OpenFlow on a consumer router and use a controller that is outside of the router's LAN. We use OpenWrt's [45] Open vSwitch [144] implementation to outsource this functionality to a cloud controller.

While it may be tempting to outsource all functionality to the cloud, a more robust solution retains some functionality locally. In particular, we continue operating a DHCP server and client locally, as well as the NAT and DNS services. This approach allows a router to safely failover to local operation when connectivity with the cloud is interrupted or if the cloud controller requires maintenance. However, even with these services running locally, our default flow management causes these flows to be elevated to the cloud, allowing it to authorize or deny DHCP leases or new NAT entries.

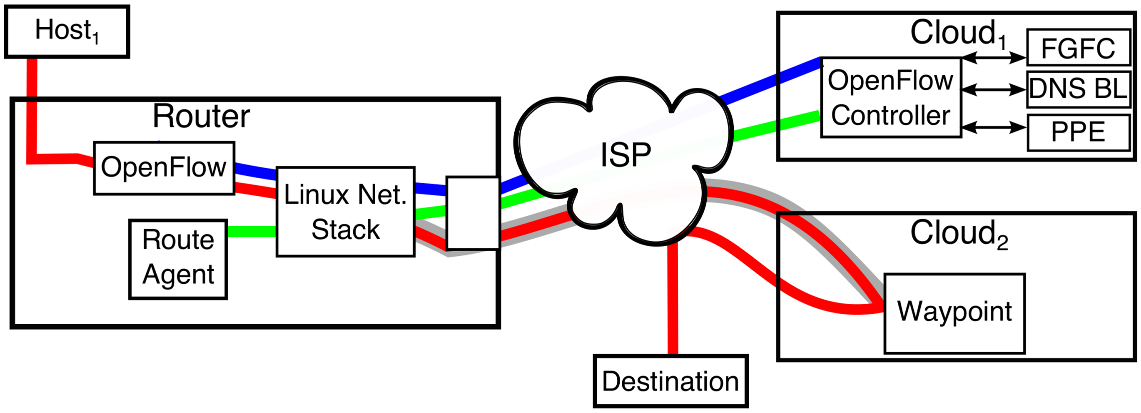


Figure 5.1: Our proposed ReSDN architecture with residential routers supporting OpenFlow. Our controller supports fine-grained flow control (FGFC) and runs a partial path encryption module that supports directing traffic to waypoints in the cloud.

In Figure 5.1, we visually depict how OpenFlow traffic is elevated to the cloud-based controller and how the commodity router directs traffic through cloud waypoints with the help of a route agent. Unlike traditional uses of OpenFlow, which may use coarse-grain rules (i.e., a rule using a wildcard for one or more fields in the flow-tuple), we focus on fine-grain rules in which the flow is fully specified, including network and transport layer source and destination identifiers. This ensures that every new flow is seen by the controller, allowing the controller to decide whether to authorize or deny each network flow as the flow initiates. Importantly, the controller may choose to divert the network flow through a proxy by requiring the router to tunnel the traffic associated with the flow through an intermediary. The route agent, which is outside of OpenFlow, can manage and negotiate the appropriate tunnels, allowing the OpenFlow controller to simply specify the targeted middlebox application, causing traffic to traverse the tunnel transparently. As we being to target more network functions, the agent will be key to deploying richer sets of middlebox functionality.

5.2.2 Cloud Controller Considerations

There are a few important considerations for the controller: its network proximity with the consumer router, the controller's software, and the approach the controller takes for managing the user's traffic. While we describe the important considerations for the controller, we note that these decisions need not directly involve the user since they can be handled automatically.

End-users may choose amongst numerous commercial cloud providers based on a variety of factors, such as cost and the computational resources provided. For a cloud controller, it is particularly important to minimize network latency. Depending on the user's geographical location and the connectivity of the user's ISP, some cloud data centers may be more appealing than others. Since the controller must be consulted at the initiation of each request, a low latency connection will minimize any delay for new flows. Software on the router can perform latency tests across a series of candidates and present the user with options, comparing likely performance with other considerations, such as cost, and let the user choose.

The user, or software acting on behalf of the user, must install and configure an OpenFlow controller. There are many options for OpenFlow controllers. We use the POX controller in our approach since it has a modular implementation that enables fast prototyping. A controller's modularity is key to enabling support for arbitrary residential network functionality, since components can be added and removed with minimal overhead.

The controller can leverage a variety of applications to make traffic control decisions. These applications can be divided into two key classes: 1) elevation-centric applications, which make decisions based only on the initial OpenFlow packet, and 2) payload-centric applications that examine all the packets in a flow. The needs, and the approach to meet these needs, differ dramatically. The elevation-centric applications can run as a traditional module in the OpenFlow controller and can make decisions based on the flow as it is elevated. Such applications may include IP-based blacklists or firewalls based on network or transport layer information. Once these applications make a decision, the controller can approve or deny the flow, completing the controller's involvement.

Payload-centric applications, such as IDS software, deep packet inspection tools or stateful firewalls, require more than a pure OpenFlow approach. In this case, the OpenFlow controller must order the consumer router to divert the flow's traffic using a tunnel to a proxying device that will inspect the flows. Importantly, this proxying device does not need to be co-located with the controller and can be selected using other criteria (such as throughput and bandwidth costs). The proxying device can inspect the traffic and asynchronously inform the controller if the flow ever needs to be terminated or modified.

5.3 Deploying Applications

We deploy two applications in a residential network setting. The first is a DNS blacklist application that acts an elevation-centric network function. The other application is a partial path encryption (PPE) approach that fits our payload-centric application profile. In our work, the OpenFlow controller performs fine-grained flow control (FGFC) to ensure all network flows are seen.

5.3.1 DNS Blacklist

The DNS blacklist is a function that the controller calls anytime a DNS request is received to determine if the domain requested is blacklisted or not. When the controller receives the DNS packet, it passes the complete packet to the blacklist function. The DNS blacklist function parses out the DNS question and compares the domain to a previously-defined list of domains. If the domain is in the blacklist, the function notifies the controller to drop the packet. Similarly, if not in the blacklist, the function notifies the controller to continue allowing other functions to process the packet.

5.3.2 Partial Path Encryption via Waypoints

Our PPE approach allows better end-to-end protection of unencrypted traffic. In many cases, online servers do not provide an encryption option. This is especially true for Internet-of-Things (IoT) devices, but even well-known sites such as `www.webmd.com` leave all communication including sensitive medical queries unencrypted. Without encryption, end-users may pass personal information such as usernames, passwords, medical conditions, or more subtle information such as political interests in unencrypted packets that are sent across in the Internet. We seek to reduce the exposure of unencrypted information with PPE via waypoints. Because we do not control the destination server, we cannot force it to encrypt traffic to an end-host. However, we may use a waypoint to encrypt traffic for a majority of a packet's routing path across the Internet, if the waypoint is strategically placed.

Figure 5.2 shows how waypoint routing helps encrypt end-to-end traffic. The bottom path from the host shows what path HTTP traffic would take if not using PPE. Without PPE, the traffic traverses 5 hops before reaching the destination. With PPE, the home router encrypts and tunnels the traffic to a waypoint hosted by a cloud provider. In the top path, the packet only traverses 3 hops, assuming a single router in the cloud provider, before reaching the destination. The number of hops saved (2 in our example) is dependent on the destination and how many hops the destination is from the nearest cloud provider.

In practice, the PPE module builds and maintains a table of destination IP addresses and two sets of hop counts to reach those destination. The first hop count is the *minimum* number

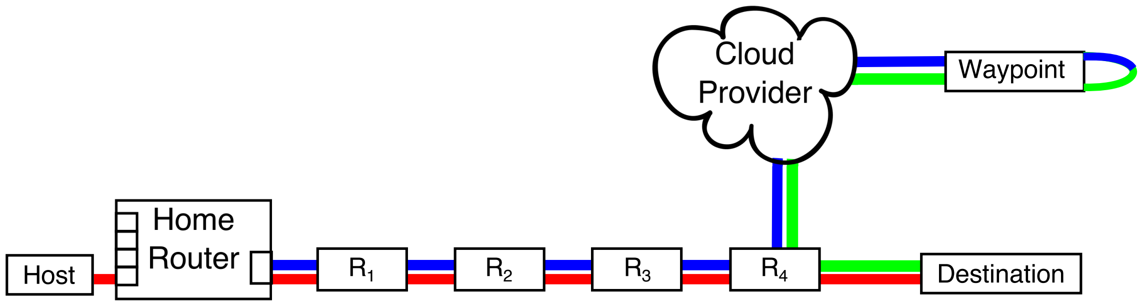


Figure 5.2: Overview of how partial path encryption with waypoint works. Traffic is encrypted at the router per flow and routed through a waypoint in the cloud where it is decrypted and routed to the destination. This incurs a smaller number of hops where the original traffic is unencrypted.

of hops required to reach a destination from each of the cloud providers that host a waypoint. The second hop count is number of hops required to reach the destination directly from the home router. The latter count can be derived by passively watching traffic and checking the TTL or explicitly generating OpenFlow PacketOut events to invoke responses from the destinations. If the hop count from any waypoint is less that that of the path directly from the residential network, the PPE module may direct the route agent to tunnel all traffic to the waypoint over an encrypted tunnel, for example, using IPSec. Importantly, this decision of using a waypoint could consider other factors such the difference in hop count or the ASes along the path since some ASes are known to be more malicious than others [159].

5.4 Implementation

To determine the feasibility of our ReSDN infrastructure, we created a prototype using a consumer-grade router and an OpenFlow controller in a cloud data center. We flashed a TP-LINK TL-WR1043ND v2 router with a custom build of the OpenWrt (Chaos Calmer 15.05) image. To enable OpenFlow support, we selected the kernel-level Open vSwitch package.

To ensure continued operation in the event of connectivity issues when reaching the cloud controller, we ran NAT, a DNS recursive resolver, and DHCP services locally along with OpenFlow. We had to create a virtual interface to act as an intermediary between the router's WAN interface and the router's internal LAN. To enable NAT functionality, we created static rules in `iptables` for masquerading. We did not have to make any special changes for the DHCP or DNS services. We can conceal the complexity of these routing configurations by including them inside the firmware upgrade process.

We then created two cloud virtual machines (VMs). One VM hosts the OpenFlow controller, and the other VM hosts a waypoint. We used Ubuntu 14.04 Linux server micro-VM instances in the Amazon EC2 compute cloud with the multi-tenant configuration, which was eligible for Amazon's free tier. Each VM has a single 2.5 GHz core with 1 GByte of RAM and used a dynamic global IP address but operates behind NAT. We performance latency tests and determine the North Virginia data center provided the best performance to our residential network. In the future, we will streamline this evaluation to enable the router to automatically determine the best cloud location.

5.4.1 DNS Blacklist

Our DNS blacklist is an elevation-centric application that consumes elevated DNS queries, which the controller naturally receives since each DNS query uses a new network flow. The DNS blacklist examines the packet, compares the requested host name with an existing list of banned sites, and drops the packet if it is on the blacklist. Otherwise, the function allows the DNS packet to be forwarded unmodified. When the controller is initialized, the blacklist function retrieves the blacklist from a remote server hosting a popular domain blacklist [55]. Because we use fine-grained flow control, packets are always elevated to the controller. Accordingly, the controller needs a way to determine if the user has enabled the blacklist or not. To do this, the controller differentiates based on the reason for elevation. If the user has not explicitly, enabled the blacklist, the DNS packet is elevated due to a flow miss. If enabled, a flow match causes elevation and allows the controller to act. We implement the blacklist as a POX [31] module.

5.4.2 PPE via Waypoints

For this work, we make a few simplifying assumptions about the destination locations. First, we only focus on popular (within top 1000 US [41]) domains that still support unencrypted HTTP traffic, which is typically transmitted using TCP port 80. Second, we use a single waypoint Amazon EC2 instance in North Virginia. Deploying more waypoints in different cloud providers and cloud locations will help provide routes closer to the destination server. Finally, we make the measurements known *a priori* the PPE module and have it update the route agent to send all traffic over the tunnel to the cloud waypoint. When a new flow connecting to a destination IP address that is known to have a smaller hop count, the module notifies the route agent over a TCP connection to direct all subsequent traffic to that IP address to the waypoint.

5.5 Evaluation

Network latency and the resulting delay introduced are the most important consideration when placing the router's OpenFlow controller in the cloud. We first evaluate the performance of the DNS blacklist function and performing TCP handshakes against a traditional router where no blacklist exists and no OpenFlow traffic is generated. We perform this evaluation using our consumer router on residential network connected via a cable modem ISP.

5.5.1 Performance Overhead Analysis

The remote OpenFlow controller introduces latency at the initiation of a connection attempt. We examine the impact of this latency in two scenarios: DNS queries and TCP connection establishment. For the DNS trials in which a controller is used (i.e., the OVS Local and OVS Cloud scenarios), we use a blacklist application on the controller that drops the DNS request if the host name is in the blacklist. We conduct 100 sequential DNS requests from a host on the residential

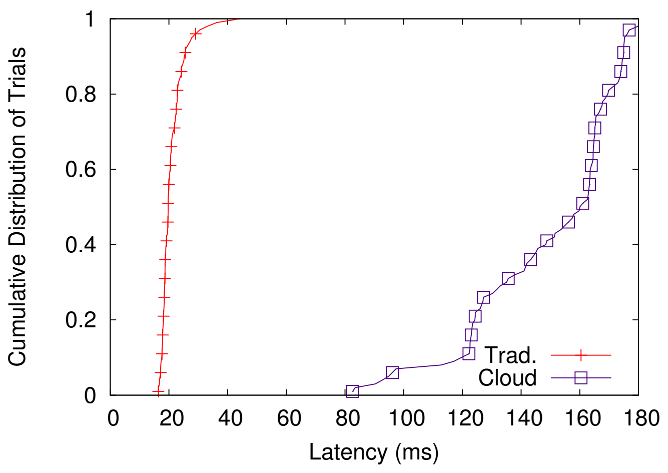


Figure 5.3: End-to-end latency of DNS requests over 100 trials.

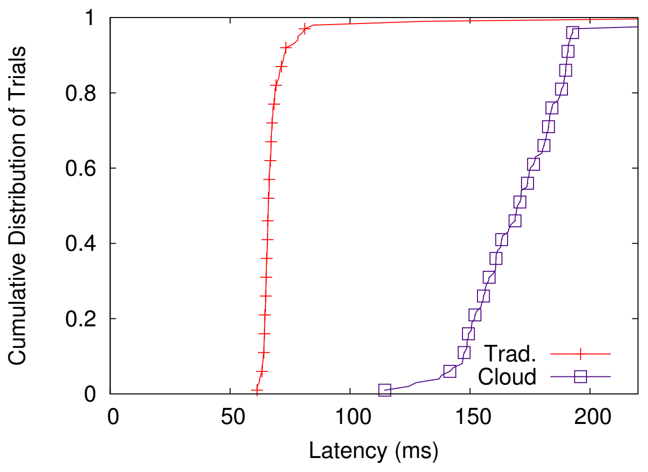


Figure 5.4: TCP handshake latency over 100 trials.

network to our ISP's DNS resolver and measure the time taken to receive the response. For the TCP experiments, we establish 100 sequential TCP connections from a host in the residential network to a geographically close server outside of our ISP's network. For each connection, we measure the time required to establish and immediately terminate the connection.

We show the results of the DNS trials in Figure 5.3 and the TCP trials in Figure 5.4. From these results, we can see that the cloud-based controller compared to a traditional network without OpenFlow adds roughly 130 milliseconds of latency to the DNS requests and roughly 105 milliseconds to the TCP connection establishment process. This latency is only incurred at the connection initiation and, from a user's perspective, it did not have a noticeable affect on the use of the network connection.

While the additional latency in these results may seem high, we found there was little impact on the actual payload transfer of a flow. Using the D-ITG [59] software package, we establish 50 simultaneous TCP connections to a remote server. We then measured the throughput, jitter, and packets per second of these connections. The home network machine sent packets between 500 and 1,000 bytes using a uniform distribution as fast as possible for 60 seconds. As expected, when examining these results, we saw no noticeable impact on jitter, throughput, or packets per second metrics. The higher latency at connection establishment in the scenarios using the controllers were essentially amortized across the network flow. We also recorded the computational overheads at the consumer router, such as memory and CPU usage, as the number of network flows increases but saw no noticeable impact across trials.

Table 5.1: Performance results for traditional network versus OpenFlow using D-ITG and 50 TCP connections. Initial connection overhead is amortized over a 60 second connection.

	Jitter (seconds)	Bitrate (Mbps)	Packets/second
Traditional	0.145	5.915	986.6
Cloud	0.147	5.926	987.6

5.5.2 Partial Path Encryption

We now briefly discuss our evaluation of the PPE module. The controller module was configured to monitor DNS responses (blacklist monitors requests) and add automatically add routes to the IP address of the domains listed in Table 5.2. For simplicity, we used GRE to encapsulate the traffic. We plan to use IPSec tunnels in follow-on work. In terms of performance, the encryption process in IPSec will add some negligible processing overhead, but we focus on the baseline network performance using GRE. In our experimental setup, the residential network was always 15 hops from the waypoint. However, when comparing the number of unencrypted hops a packet will take to the destination, these encrypted hops are excluded from the total since the tunnel will protect the packet over these hops.

Table 5.2 shows the two domains we used in testing our PPE module. One domain was within the same Amazon datacenter which provides the absolute best case of only traversing a single router within the data center. The other domain was 17 hops from the data center. However, using a cloud waypoint for both domains, allows us to reduce the number of hops outside of our tunnel by 14. We also show the average RTT found by retrieving each Websites `index.html`. The RTT differences includes the initial packet elevation which causes a slight increase in the average.

Table 5.2: PPE hop reduction between the residential network and an Amazon EC2 waypoint and the RTT to the destination.

Domain	Hop Count			Round Trip Time (ms)		
	Waypoint	Direct	Diff.	Direct	Waypoint	Diff.
`www.sbnation.com`	1	15	-14	2.13	3.91	+1.78
`www.mentalfloss.com`	17	31	-14	1.14	3.60	+2.46

Finally, we ensured throughput was not noticeably degraded using the waypoint. Large transfers were not readily available from the websites we tested. Accordingly, we download a 100MB file both directly from the residential network and through the waypoint using PPE. In both instances, we had an average throughput of approximately 2.2 Mbps. We further explore throughput measurements using cloud proxies in Chapter 8.

5.6 Conclusion

In this Chapter, we proposed using a cloud-based controller and network function virtualization to enable enterprise middlebox services in residential networks. We created a prototype system on a consumer-grade router and found that outsourcing functionality to the cloud yielded overheads that were imperceptible to a user. In doing so, we have highlighted the potential for arbitrary cloud-based controls for residential networks.

Chapter 6

ReSDN Application 1: Whole Home Proxies

6.1 Introduction

In Chapter 5, we present our ReSDN infrastructure and discuss two relatively simple applications that have security benefits. However, our infrastructure can be used for other more complex classes of applications. One example class of applications is that of a "whole home" proxy solution that is tailored to specific applications. Enterprises often employ proxies to detect and block access to potentially malicious destinations or content. By employing this protection at the perimeter, enterprises can provide protection to many hosts at once. This goal is shared by residential networks. In particular, residential networks have numerous heterogeneous devices, including desktop and laptop computers, mobile devices, and embedded devices, including televisions, receivers, and video game consoles. Some devices, particularly for mobile or embedded devices, may not have options to allow users to configure proxy settings or other advanced networking features.

Residential users have limited options for a whole home proxy solution. Many commodity routers lack options for setting up proxy servers or site-to-site VPN end-points in their manufacturer-provided firmware. Even if users replace their routers with high-end devices or install custom after-market firmware (which can be daunting even for technical users [112]), the controls are too coarse grained. Many VPN setups allow the tunneling of all network or none at all. "Split tunnel" VPNs can allow partial rerouting of traffic, but those tunnels are created on a per-destination basis rather than on a per-flow basis. Finally, the complexity of managing these VPN tunnels may be cumbersome for users.

We propose to change the network model. Rather than require home users to become experts, we focus on outsourcing security management to expert service providers. We explore modifications to commodity residential routers to allow them to export management to a remote controller, using the OpenFlow protocol [129], and a series of device proxies. Unlike traditional OpenFlow, we will examine the payload of network traffic and use remote cloud nodes to protect residential users.

In exploring this concept, we focus on the Skype video conferencing application. Skype is commonly used, with over 300 million users worldwide [36], with support on devices ranging from computers to mobile devices and video game consoles. Skype uses a peer-to-peer connection between communicating parties which can reveal the IP address of a Skype user to others, whether they are aware of an established connection or not [122]. Some blackmarket providers offer to denial-of-service attack users when provided with a target Skype username since the Skype directory service leaks the IP addresses of connecting parties [38]. We believe Skype is a particularly good example application because it is known for breaking through common network barriers (like firewalls), uses a proprietary protocol that cannot be altered, and has complex infrastructure. Simply put, a technique that works for Skype will likely work for many simpler network applications.

In pursuing this work, we make the following contributions:

1. **Proxies on a per-flow basis:** We combine an OpenFlow approach with proxies and a tunneling agent on the router to proxy communication on a per-flow basis.

2. **Demonstrated utility of an application-specific proxy:** We create an SDN controller application, agent at the router, and proxy configuration for Skype that demonstrates the viability of per-flow proxies that are tailored to applications in a residential network.

3. **Evaluated the performance and effectiveness of the approach:** We evaluate the approach using 5 different devices running Skype on a home network with a cloud-based OpenFlow controller and proxy.

6.2 Related Work on Skype

Our approach relates to work in detecting if a network flow is related to the Skype protocol and measures to try to influence Skype privacy. We now describe research surrounding Skype and privacy related research.

6.2.1 Distinguishing Skype Network Traffic

Skype has a complex peer-to-peer (P2P) infrastructure with supernodes (which are used for routing), ordinary nodes (such as end-user machines), and a login server [60]. Many researchers have tried to characterize and understand how the underlying Skype protocols work [60, 146], while others have focused on detecting Skype traffic in networks [69, 70, 142].

In this work, we do not attempt to decode Skype's proprietary protocol to fully determine how Skype learns or transmits the IP addresses of the communicating peers, so we must simply detect and proxy all messages associated with the Skype program to ensure the user's real IP address is not leaked. SkyTracer [180] has a similar goal of detecting Skype traffic at the flow-level. SkyTracer uses a mixture of flow tuple and byte-level packet characteristics to identify Skype traffic within the first few packets. While such approaches may work well for identifying ongoing or new Skype calls, we must be able to detect Skype activity *before* the associated network traffic leaves the network. In particular, we must be able to proxy all communication to Skype servers, supernodes, and ordinary nodes to avoid revealing the user's IP address.

6.2.2 IP Address Privacy in Skype

Since Skype uses a P2P connection to directly establish a connection between communicating hosts, each host naturally learns the IP address of its communicating counterpart during a call. However, Le Blond *et al.* [122] describe a method to passively obtain the IP addresses of thousands of Skype users without alerting the user. They further describe linking a user's IP address to other Internet activity such as BitTorrent traffic. While Le Blond propose infrastructure changes, their approach does not completely address the issue. The Skype client (SC) application could simply only allow added contacts to establish direct connections; however, this is only enabled on the iPhone and not any of the other devices we tested. The Xbox One likewise only allows immediate contacts to connect, but it does so without determining whether a connection is direct or not. These features could easily be undermined with a social engineering attack in which the attacker is added as a contact.

In other work, Ehlert *et al.* [88] found that even when manually configuring Skype to use a proxy server in the client's settings, Skype will still try to establish a direct connection with the peer and will only use the proxy as a last resort if the earlier efforts fail. As a result, users may believe they are masking their actual IP addresses behind a proxy only to have Skype bypass the proxy and establish a direct connection.

6.3 Approach: Tailored Proxying and Tunneling

A user may run many programs, each with their own workflow and associated security concerns and goals. To ensure these security goals are met, we enable security experts to write tailored control applications to manage the network traffic of the user's applications. We then create a general platform and API that allows those experts to run their control application across many different types of residential networks.

Our general platform consists of four components: a commodity residential router running custom firmware, a cloud-based OpenFlow controller that directs the router's behavior, a cloud-based proxy/middlebox that monitors traffic, and a GRE tunnel between the router and proxy. These components are common across applications and services. To tailor the system to a particular user program, a security expert will create an custom application on the OpenFlow controller to manage the features. Further, the expert may run custom software on the proxy/middlebox to enforce these goals.

We instantiate this general approach with a specific application for the Skype video conferencing application. We now describe each of the components in the general platform and the customizations needed to meet Skype's security goals.

6.3.1 Platform: Router, Controller, Proxy

We modify a consumer-grade router to support the OpenFlow protocol by installing the OpenWrt's [45] firmware and enabling the Open vSwitch [143] module. Unlike prior work, we control the router remotely with an OpenFlow controller that is hosted at a cloud provider. This controller has the ability to vet all of the new connections established through the router, including traffic within the LAN and Internet traffic. The router establishes a connection to the controller upon boot and requests instruction for new network flows.

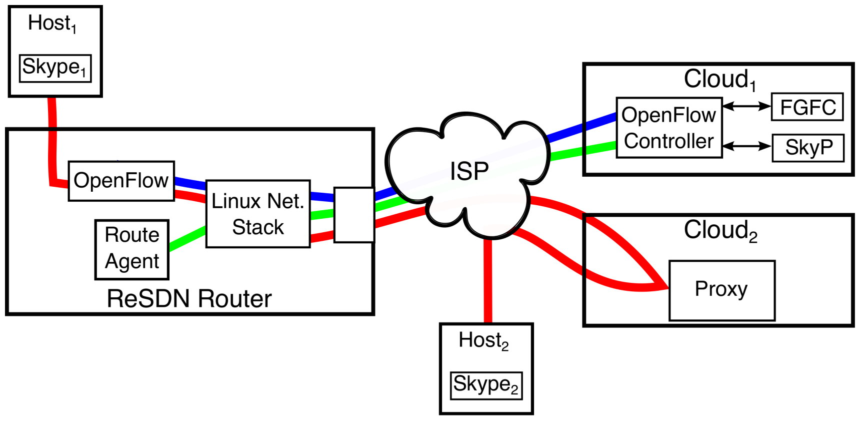


Figure 6.1: Overview of how our Skype proxy approach works using multiple cloud providers for controlling OpenFlow and proxying traffic. Our controller uses fine-grained flow control (FGFC) and the Skype Proxy (SkyP) module to detect Skype calls and update the route agent to send traffic through the proxy.

We then create a cloud virtual machine that operates as a middlebox or proxy server. In its most basic form, the proxy simply uses network address translation (NAT) and forwards packets from the consumer's router to the requested destination and vice versa. To facilitate communication with the proxy, the router creates a GRE to a list of eligible proxies upon booting. When ordered to do so by the OpenFlow controller, the router simply uses the appropriate GRE tunnel as the destination for selected flows, causing them to be sent via the proxy.

6.3.2 Tailored Control: The SkyP Module

While the basic platform provides a mechanism to send arbitrary traffic via a cloud-based proxy, there must be a module or application that indicates which traffic should be sent to the proxy and what the proxy should do with the traffic once it receives it. This module may be different for each type of application protocol to provide tailored control.

For Skype traffic, we create a custom controller application, which we call the *SkyP Module*. This module uses Skype network characteristics to detect what traffic is likely associated with Skype and directs that traffic via the proxy. Since Skype is a complex proprietary protocol, we do not know which messages are used to register the user's IP address in the Skype directory. To prevent the user's real address from being leaked, our SkyP Module must take a series of steps to determine what traffic is Skype-related.

There are two main features the SkyP module must consider: 1) communication with known Skype infrastructure or 2) direct P2P communication. We use DNS features and IP ownership to identify and proxy the connections to the Skype infrastructure. However, for P2P communication, we leverage the fact that the client initiates its P2P connections using a randomly-generated port number that is created upon installation of the client. Using the approaches described in the following sections, we can learn the client's P2P source port. Once we have done so, we watch for any peers the client contacts using the P2P source port and proxy all traffic to those discovered peers (since traffic subsequent to the rendezvous may communicate using random ports). We now describe each of these approaches in features in detail.

Skype DNS Requests

When the Skype client (SC) first starts, it initiates a series of DNS requests to hardcoded domain names that are included as part of the Skype executable. Some DNS host names, such as `ui.skype.com`, are fixed while others, such as `dns13.d.skype.net`, appear to be members of a load balancing group that the SC may rotate amongst. To create a complete list of DNS host names associated with Skype, we examined a diverse set of devices and operating systems as shown in Table 6.1.

Table 6.1: List of devices used in our experiments

Device	Operating System	SC Version
iPhone	iOS 8.4	6.1.0.210
Macbook Pro	OS X 10.10.5	7.10
Dell Laptop	Windows 7	7.10.0.101
Del Laptop	Ubuntu 14.04.3	4.3.0.37
Xbox One	Xbox OS 6.2.13332.0	1.9.0.1003

For each device, we launch the SC, initiate a roughly five second long voice call, and close the SC. We repeated this process 20 times for each application, flushing the device's DNS cache. In performing these trials, each client was behind a NAT device since Skype is known to exhibit different behavior when operating behind NAT [60].

From these trials, we created a list of 32 host names that appeared to be related to Skype. Of the 32 hosts, 6 had distinct patterns that could be generalized into a regular expression. For example, there are 18 different host names that match the pattern `dsn[0-17].d.skype.net` [146], allowing us to easily construct a regular expression to match the hosts. We augment the list of addresses we empirically discovered with the list of important host names discovered in prior work [146].

We configured the SkyP module to monitor DNS requests for these host names. Since the SkyP module can receive all packets elevated to the OpenFlow controller, including DNS packets, it can analyze these requests and their responses. Each time a client initiates a DNS request, OpenFlow controller sends a copy of the DNS response to the SkyP module. The SkyP module then parses the DNS response packet looking for replies for any of these known host names. If one is detected, the SkyP module extracts each of the IP addresses contained within the response. The SkyP module then directs the router to send all traffic to those IP addresses through the proxy.

Skype's Use of NAT-PMP

When the SC is first installed, it randomly generates a port number that it will use when it later attempts to create P2P connections [60]. To facilitate communication even through NAT middleboxes, the SC uses the NAT Port Mapping Protocol (NAT-PMP) [74] to request that certain ports be mapped to the SC via the NAT device's public IP address. By simply monitoring for these NAT-PMP requests, which are elevated to the SkyP module, we can learn what port the SC uses for P2P connections and subsequently direct any traffic originating from the host using that port to be sent via the proxy. Since other unrelated applications could also initiate NAT-PMP requests, we only learn ports from NAT-PMP if they are within a delta of 4 seconds of a SC-related DNS request. This approach was effective for each of the devices in Table 6.1 across 80 call sessions, excluding the Xbox One (which does not use NAT-PMP).

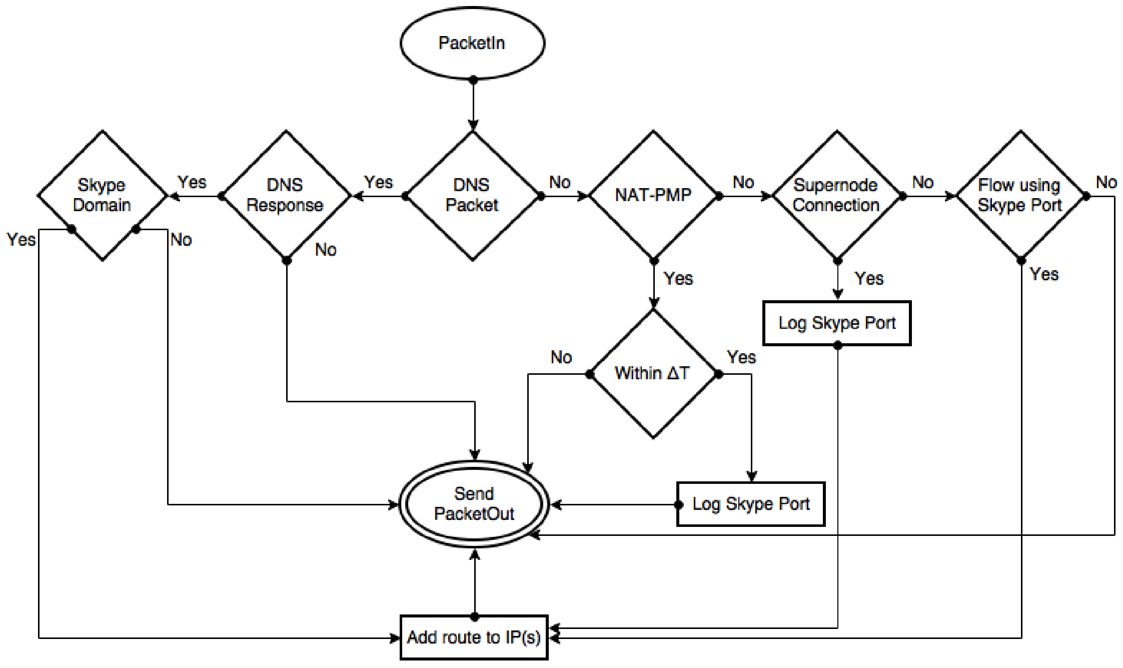


Figure 6.2: Our decision-making process in the SkyP module for proxying traffic.

Skype's Interactions with Supernodes

Since some devices, such as the Xbox One, do not use NAT-PMP, we use another SC characteristic to learn the SC's P2P port. When started, the SC makes multiple connections to supernodes. The first such connection uses the SC's dedicated port. Accordingly, by knowing the identity of all supernodes, or features associated with those supernodes, we can watch for any connections to those supernodes to learn the SC's P2P port. Prior work found that connections to supernode IP addresses typically use the port range 40001-40047 [146]. Further, all supernodes are now operated by Microsoft [35] [37], so we can examine whether the destination IP address belongs to Microsoft-owned IP space to determine if the connection is to a supernode.

6.4 Implementation

We implement our approach using a consumer-grade router and elevating flows to a remote OpenFlow controller on a server in a cloud data center. We flash a TP-LINK TL-WR1043ND v2 router with a custom build of the OpenWrt (Chaos Calmer 15.05) image. To enable OpenFlow support, we selected the kernel-level Open vSwitch package.

To ensure continued operation in the event of connectivity issues when reaching the cloud controller, we ran NAT, a recursive DNS resolver, and DHCP services locally along with OpenFlow. We had to create a virtual interface to act as an intermediary between the router's WAN interface and the router's internal LAN. To enable NAT functionality, we created static rules in `iptables` for masquerading. We did not have to make any special changes for the DHCP or DNS services. In a production deployment, these complex routing configurations would be concealed from users by including them inside the firmware itself.

We then created two cloud virtual machines (VMs) to host the OpenFlow controller and anonymizing proxy. Each VM was an Ubuntu 14.04 Linux server micro-VM instance in the Amazon EC2 compute cloud and was eligible for Amazon's free tier. Each VM has a single 2.5 GHz core with 1 GByte of RAM and uses a dynamic global IP address. We ran a script to install and launch the POX OpenFlow controller with our own fine-grain flow control and SkyP modules.

The anonymizing proxy is configured it to implement a source NAT using `iptables`. With this configuration, the proxy automatically translates and forwards traffic to and from the GRE tunnel connected to the home router. It only performs network-layer translations, so the Skype P2P port will be exposed in network communication. We did not explore performing port address translation at the proxy.

6.5 Evaluating SkyP

We evaluated our approach by performing Skype voice calls and verifying functionality using third-party Skype IP address lookup applications, such as Skype Resolver [38], and via Wireshark captures. The specific devices and software versions are listed in Table 6.1.

6.5.1 Evaluation Setup

Our evaluation setup is shown in Figure 6.3. We position the client using our whole home proxy in a residential network behind NAT. In our evaluation, the call initiator and responder are already contacts.

For each device except the Xbox One, we performed the following steps. First, the SC using SkyP (Host_1 in Figure 6.3) attempted a voice call to Host_2. After establishing the call, Host_1 sent a chat message, transmitted an image file 3 MBytes in size and ended the call after approximately 2 minutes. We then had Host_2 attempt a voice call to Host_1 to ensure the proper IP address was used to establish the P2P connection through the proxy. The Xbox One's version of Skype does not allow chat or arbitrary file transmission. As such, we only performed VoIP calls with the gaming console.

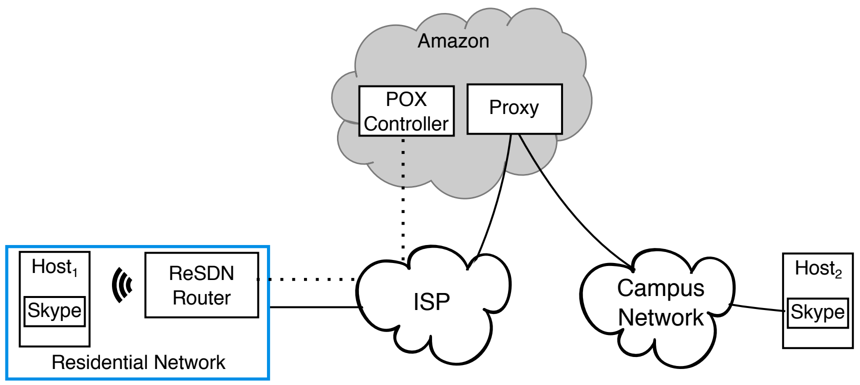


Figure 6.3: Our evaluation setup for testing SkyP on different devices.

6.5.2 Verification

We verified our approach works in two ways. First, we used an online third-party tool, Skype Resolver [38], to ensure the only IP address associated with our username was the IP address of the proxy. Because hosts can be associated with multiple IP addresses at once, such as a mobile device and an office computer, we waited to perform the experiments until no other IP addresses were cached.

We performed packet captures at each host to verify correct proxying. For each device being tested, we verified each packet capture individually to ensure our IP address was never leaked to Host$_2$. We observed that all VoIP call, chat, and file transmission traffic established connections to Host$_2$ using our cloud proxy or were transmitted via an anonymizing supernode (for chat and file transmission). The Skype Resolver only learned the proxied IP address; it was never able to detect the real IP address of the proxied user.

6.6 Discussion

In evaluating our Skype setup, we found an interesting edge case. When the two communicating parties are not already contacts in the Skype system, a direct connection can occur if the adversary uses an unrestricted publicly routable address. In this case, the adversary sends a request through the Skype supernode to the internal host. That request causes the internal host to directly connect to the adversary. This particular workflow bypasses DNS, NAT-PMP, and supernodes and thus we do not proxy the connection correctly.

This approach, of requesting the other party to initiate the connection, is particularly useful for Skype to bypass NAT. Since one of the machines uses a publicly routable address, it can act as a server to have the other machine connect. By sending a request to this effect via the Skype supernode, the machines establish a connection.

Since the Skype protocol is encrypted, we cannot detect the IP address for these new requests and simply proxy all connections to that IP address. However, the connection request packet appears to use a packet size in the range of [329-339] bytes. As a workaround, we add a function not shown in Figure 6.2 to proxy any new network flows that occur within 200 milliseconds of these requests. As such, the requirement of peers being pre-existing contacts is no longer necessary.

6.7 Conclusion

We proposed an approach that uses network function virtualization to enable a "whole home" proxy for residential networks. Using a cloud-based controller and proxy, we are able to control traffic on a per-flow basis that is immediately deployable. Using Skype as a motivating example, we found that even a complicated proprietary protocol can be singled out and selectively proxied. In doing so, we have highlighted the potential and discussed other applications for application-specific cloud-based proxies in residential networks. In future work, we will explore other types of proxies, including IPSec and caching proxies.

Chapter 7

ReSDN Application 2: Validating Security Protocols

7.1 Introduction

Residential networks have high device diversity, including more traditional systems such as laptops or desktop computers, but also include Internet-enabled televisions, video game systems, and home automation systems. These embedded devices, sometimes called Internet of Things (IoT) devices, may have vulnerabilities that go unaddressed, either by the manufacturer or the end-user [104,138].

Our goal in this Chapter is to protect residential devices by ensuring the authenticity of the communication between the devices and outside systems. Essentially, if we can protect devices from communicating with untrustworthy third-party systems, we can prevent devices from being attacked. In several security protocols, such as Transport Layer Security (TLS[1]), SSH, and IPSec, the initial connection negotiation phase has the greatest vulnerability [71], since it requires confirmation of the other party's authenticity. Given the prominence of TLS in web security and online protocols, we focus on this protocol as a concrete example and later discuss how the approach can be applied to other protocols.

In this work, we ask three research questions: 1) To what extent can we perform in-line TLS certificate verification and revocation validation using cloud-based middleboxes? 2) How can we minimize the performance impact of cloud-proxying on long-lived network flows? 3) To what extent can SDN middleboxes provide novel support for other important security protocols?

In performing the work, we make the following contributions:

- **Implementation of a Novel Cloud-Based TLS Validator:** We created a new verifier, called *TLSDeputy*, that monitors the TLS handshake process and performs independent verification of TLS certificates and revocation checking using certificate revocation lists (CRLs). Such revocation checks were particularly important following the recent HeartBleed vulnerability [87].

- **Evaluation of the Cloud-Based Validator:** We verified the efficacy of the TLSDeputy across diverse devices and showed that it could increase device security. In particular, we showed that the TLSDeputy prevented smartphone web browsers, which are known to not properly check for certificate revocations [124], from reaching untrustworthy web sites. Our

[1]TLS is the successor protocol to Secure Sockets Layer (SSL).

approach behaves similar to a client performing full-chain TLS verification and revocation check and can feasibly be used today. Finally, we evaluated the tool across 40,000 top web sites and found that it properly determined which HTTPS servers were valid and which were not, demonstrating its real-world viability.

- **Created a Novel Communication Channel for the Middlebox:** By embracing the concept of participatory networks [92], we created a new communication channel between the OpenFlow SDN controller and the cloud-based middlebox. In doing so, we were able to migrate a network flow to use a direct path from the user's network after TLSDeputy confirms the TLS handshake was proper and authentic. This addresses known limitations of MB and controller consistency [90, 99].

7.2 Transport Layer Security Background and Related Work

Given our emphasis on TLS as a working example, we provide a background on the protocol and on work that aims to improve the protocol. We then describe work for using SDNs to outsource residential network security.

7.2.1 TLS Background

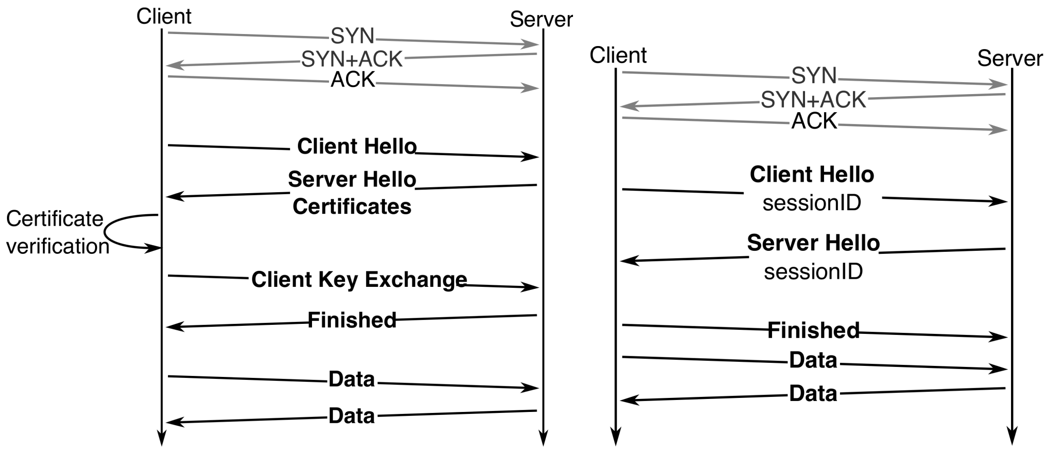
All TLS connections are preceded by a TLS handshake in addition to a TCP handshake. Figure 7.1(a) shows a full TLS handshake[2] where the server provides the corresponding certificate chain. That certificate chain starts with a self-signed, well-trusted root certificate. The root certificate signs the next certificate in the chain, attesting to that certificate's validity. The process continues with each certificate signing the next one until the process concludes at the server's own certificate.

Upon receiving the certificates from the server, the client then verifies each certificate in the chain. After verification, the client and server create a session key to use for encrypting the data to be transmitted. As a performance enhancement, Figure 7.1(b) shows how future TLS connection establishment from the client can be abbreviated by transmitting a session ID that is cryptographically derived from a previous handshake. Since certificate verification happens early in the communication between the client and server, our approach can ignore the remaining TLS connection once the certificates are successfully verified.

7.2.2 TLS Research

Researchers have performed Internet-wide scans and those of the Alexa top 1 million [6] domain names in recent years. Holz *et al.* [109, 110] have performed multiple investigations of TLS certificates to determine characteristics such as error codes in verification, chain length, and ciphers. Zhang *et al.* [182] performed scans in response to the devastating Heartbleed [87] attack. This attack, and the subsequent analysis, shows the importance of a certificate revocation system. However, few end-hosts check for the actual revocation status of certificates even a year after that

[2]Clients may also authenticate to the server but we exclude this case from our discussion.



(a) Full TLS Handshake (b) TLS Renegotiation

Figure 7.1: TLS Handshake

attack. Liu *et al.* [124] found that, with the exception of Extended Validation (EV) certificates, there is wide-spread failure in desktop web browsers to check certificate revocation lists for revoked certificates and no mobile platform browsers did so.

TLS is vulnerable to man-in-the-middle (MITM) attacks when clients fail to properly verify certificates or when malware has installed new root certificates. Recent attacks have demonstrated the ease of deploying MITM attacks on some embedded devices [104]. Dacosta *et al.* [80] provide an efficient approach to detecting MITM attacks by allowing domain servers to use a previously-established, secure channel to provide additional information to directly vouch for certificates. Huang *et al.* [113] detect live MITM attacks by detecting forged certificates through a browser Flash application.

To prevent SSL attacks such as a MITM, researchers have used various methodologies for improving overall security. Georgiev *et al.* [101] found vulnerabilities in several security-critical applications and attributed the problem to application developers misinterpreting SSL library APIs. SSLint [105] was built as a static analysis tool that will detect applications that are misusing SSL APIs. Frankencerts [71] is a blackbox solution that automates the vulnerability detection process in SSL libraries by generating certificates to test for certain vulnerabilities. Our work is orthogonal in that TLSDeputy detects and prevents insecure connections.

Client resource and performance limitations have led to a number of research efforts. Server-Based Certificate Validation Protocol (SCVP) [95] is an approach to enabling clients to delegate path construction and certificate validation to another server. This proposed standard has similar goals to TLSDeputy but requires client-side support, which may not be feasible in legacy or embedded devices. Naylor *et al.* [137] broadly quantifies the performance costs associated with deploying HTTPS over HTTP, which includes additional latency and inability to effectively use caches. Zhu *et al.* [183] more specifically focuses on the performance associated with OCSP. While their work shows OCSP response times are getting better, Liu's [124] work shows that CRLs are still the most popular revocation process for all certificates (leaf and intermediate CAs). For example, less than

50% of intermediate certificates support OCSP as compared to 99% that support CRLs.

7.2.3 Existing TLS Security Systems

Some browsers are taking steps to improve revocation checks. Chrome has introduced CRLSets [10] that contain an internally maintained list of CRLs. Which CRLs are included is not publicly known. However, the total size is limited to 250KB. Similarly, Firefox is beginning its own approach called OneCRL [34]. In contrast, our work actively maintains a large CRL database that does not need to compromise between CRL size and security.

ICSI Notary [17] is a system that passively collects certificates from participating gateways. Clients can perform DNS queries using a hashed digest of a certificate to the Notary. The DNS response contains information based about the certificate based on what the participating gateways have observed. The ICSI Notary's does not provide an enforcement mechanism but could provide another reference point for TLSDeputy's certificate validation.

Finally, Barracuda [8] has developed hardware to provide an inline application firewall that will maintain CRLs and perform OCSP checks for client certificates. That approach only focuses on revocation (no verification) and only for client certificates, which are rarely observed within the residential environment.

7.3 Securing Connection Establishment

Our goal is to protect applications that conduct important security interactions at the beginning of a connection. As part of our running example, we show how our work supplements traditional TLS verification and provides a practical approach to enforce certificate revocation checks.

7.3.1 System Overview and Trusted Computing Base

Our system uses OpenFlow-enabled switches, cloud-based controllers and middleboxes, and custom OpenFlow agents (OFAs). In Figure 7.2, we show an overview of our system with logical OpenFlow protocol communication depicted using dotted lines.

We consider all the cloud infrastructure, including the middlebox and OpenFlow controller, along with the ReSDN router to be within our trusted computing base.

7.3.2 Cloud-based Flow Redirection

Our approach requires that some network traffic be inspected by MBes in the cloud. We use an OpenFlow controller and residential routers that support OpenFlow to redirect network flows as needed. Without a connection to an OpenFlow controller, our switch acts as a Layer 2 learning switch and mimics the behavior of traditional residential routers. That is, all required services for an Internet connection such as DHCP, DNS, and NAT all function without being connected to an OpenFlow controller. This allows us to safely fail-over in the event the OpenFlow controller goes offline. When connected, our OpenFlow router enforces fine-grained flow control. Any new network connection resulting in a new network 5-tuple (IP$_{src}$, IP$_{dst}$, Port$_{src}$, Port$_{dst}$, transport protocol)

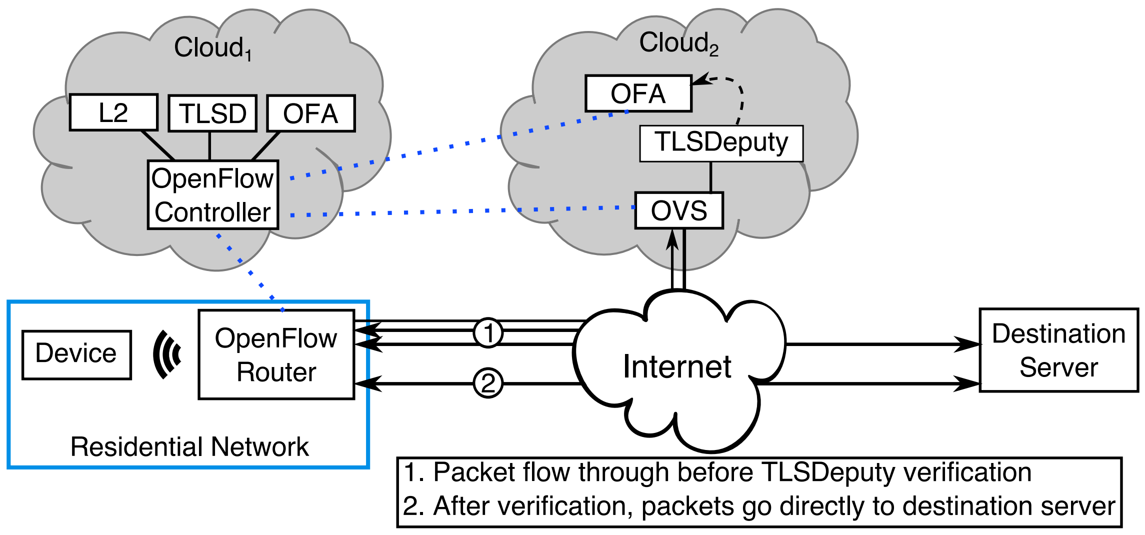


Figure 7.2: Our system uses a cloud-based OpenFlow controller and middlebox for TLS verification and revocation. TLSDeputy relays verification results to a controller module using a special OpenFlow Agent (dashed line). Similarly, TLSDeputy has a module on the controller that steers new TLS connections through its MB software. Blue dotted lines represent logical communication using OpenFlow.

will require approval from the controller. The controller can then use packet-level information at the start of a connection to determine how the flow should be handled and whether or not a MB service is required.

By default, our controller performs basic Layer 2 learning to forward packets. In addition to Layer 2 learning, our controller runs a module to detect new TLS connections (labeled TLSD) and an OFA module that will communicate with the OpenFlow agent on the MB. When the TLSD module detects a new TLS connection, the TLSD module instructs the controller to send OpenFlow FlowMods to the Open vSwitch (OVS) instance in the cloud and to the home router. Those FlowMods will cause the router to tunnel all incoming and outgoing TLS packets through the cloud MB. These rules ensure that the MB will see the bidirectional communication between the client and TLS server.

The loopback communication path from the cloud MB, shown in Figure 7.2, allows us to remove the loop once the TLSDeputy has verified the TLS handshake. This restores the performance benefits of direct communication without the MB. If we instead proxied the connection through the MB, we would not need the loopback technique but would also never be able to migrate the connection away from the MB without breaking the end-to-end connection.

7.3.3 TLSDeputy Middlebox

Our TLSDeputy middlebox runs within a cloud VM that is connected to an OVS instance. The TLSDeputy monitors the TLS handshake and checks certificates and other important information, such as the Server Name Indicator (SNI) extensions to TLS, to ensure a secure TLS connection. In addition to checking for certificate revocation, TLSDeputy performs certificate verification and other similar tasks that the end-host also performs. We provide TLSDeputy with a trusted root certificate store containing 180 root certificates that were extracted from Mac OS X 10.11.3 to allow the TLSDeputy to verify certificate chains.

The OpenFlow controller detects TLS traffic using transport layer ports and diverts all TLS traffic to the TLSDeputy beginning with the TCP SYN packet. The TLSDeputy inspects the Client and Server Hello messages. First, the TLSDeputy checks to see if the TLS request is a renegotiation or a new connection. If both the client and server transmit a Session ID value in their handshakes, TLSDeputy recognizes the connection is a valid renegotiation and notifies the OpenFlow controller via the OFA that the communication can be transmitted directly via the residential router without further TLSDeputy inspection. Otherwise, TLSDeputy knows the connection is a new negotiation and performs detailed verification checks.

If the client uses the SNI extension and specifies a server's host name, e.g., `www.example.com`, in the Client Hello message, we store that value to later verify the host name in the server's certificates. Next, the server responds with a Server Hello and immediately sends certificates, as shown in Figure 7.1(a). TLSDeputy parses the server's response and extracts each certificate being provided. As per RFC 5246 [82], the first certificate in the chain is the destination server's certificate. The subsequent certificates are then ordered such that the preceding certificate is directly certified by the next. The chain terminates, optionally, with the self-signed root certificate. Since TLSDeputy only trusts the root certificates that are pre-loaded in its local store, it ignores any self-signed root certificates.

Once the server sends the last certificate in the chain, TLSDeputy performs its verification before allowing the connection to continue. TLSDeputy passes the certificates and the client's indication of the server's host name, if any, to the verification submodule. For our verification, we use LibreSSL [22], which is a hardened implementation of the popular OpenSSL library. Since relatively few client implementations use LibreSSL currently, TLSDeputy's use of LibreSSL provides software diversity which may yield more robust security. We convert each certificate into a corresponding X.509 standard certificate data structure and store the certificate. We use our root certificates to verify each of the provided certificates.

After completing the verification, TLSDeputy removes the flow from consideration and releases the remaining associated packets. TLS deputy can then watch the client's response to the packets. If a device proceeds with the connection when TLSDeputy found verification issues, TLSDeputy will detect the device is improperly verifying TLS handshakes and will break the connection. Optionally, the software can notify the user of the issue.

7.3.4 CRL Enforcement

Before the TLS certification chain can be verified, we must determine what CRL checks to perform. Due to implementation details in both LibreSSL and OpenSSL, there are only two options: only verify the server's certificate or verify the entire chain. If any certificate in the chain lacks a CRL (i.e., lacks a URL where the CRL can be obtained), we cannot perform a full chain verification. Likewise, if the server's certificate lacks a CRL, no CRL verification is possible.

One of TLSDeputy's most important functions is to provide an approach that allow for efficient full path CRL enforcement. Recent work [124] has shown that no mobile browser performs revocation checks even after the high-profile Heartbleed attack. Liu *et al.* speculate that performance is likely a contributing factor given that their Internet-wide scan found the weighted average CRL size to be 51 KB. The size of CRL becomes more concerning as the length of the certificate chain grows. The average length of a valid chain has been shown to be 2 (a single intermediate CA) [57]. TLSDeputy addresses these concerns by proactively caching CRLs locally rather than obtaining them on demand.

To determine which CRL to consult, we check the CRL distribution point extension in each X.509 object. For each certificate, we retrieve all the available the URIs distributions points[3] provided. Beginning with the server's certificate, we iteratively check for revocation using each certificate's indicated CRL. If we have successfully retrieved CRLs for all certificates in the chain, we perform a full-chain CRL check with LibreSSL.

7.3.5 Enforcing TLS Validation via Participatory OFAs

The TLSDeputy can be more efficient with assistance from the OpenFlow controller. If the TLSDeputy can communicate TLS verification information to the controller, the controller can then allow subsequent packets in the connection to be routed directly (if TLSDeputy verification passed) or install a drop rule at the residential router (if TLSDeputy verification failed). This optimization is an example of the "participatory networks" concept. Essentially, the OpenFlow controller enforces policy in the network yet relies upon MBes to perform detailed inspection that is not feasible at the controller. However, traditionally, the controller and MBes cannot share information and collaboratively enforce policy.

Others have attempted to address the problem of SDN and MBes by modifying packets in-flight to hold additional information. For example, FlowTags [90] overloads the 6-bit Differentiated Services field in the IP header of a packet to pass information between OpenFlow switches. OpenMB [99] suggests making the internal state of a MB accessible to the controller to allow the controller to understand what actions were taken. These approaches are limited in the amount of information they can share or in the amount of redesign necessary for support. To address this problem, we embrace the notion of participatory networking [92] whereby MBes can relay information to the controller to enable flow-level decisions. FRESCO [156] has a similar notion of enabling an API where MBes can send information out-of-band to their applications. In contrast, our approach, shown in detail in Figure 7.3, allows a MB to embed arbitrary information into an OpenFlow PacketIn message and transmit that in-band to the OpenFlow controller.

Although the middlebox communicates using an OpenFlow PacketIn event, the payload of that message uses a custom payload recognizable only by our own specific OpenFlow controller module. We configure the controller so that the only module listening for events from the OFA is the OFA module that we designed for this purpose. Accordingly, we can pass any arbitrary information to the module relating to MB state. In our work, we pass the flow tuple when verification has finished, the status (e.g., success or failure) and an additional message describing the reason for failure, if appropriate. Future work will integrate this approach with other MB applications such

[3]We ignore unreachable distribution points such as `ldap://` and `file://`.

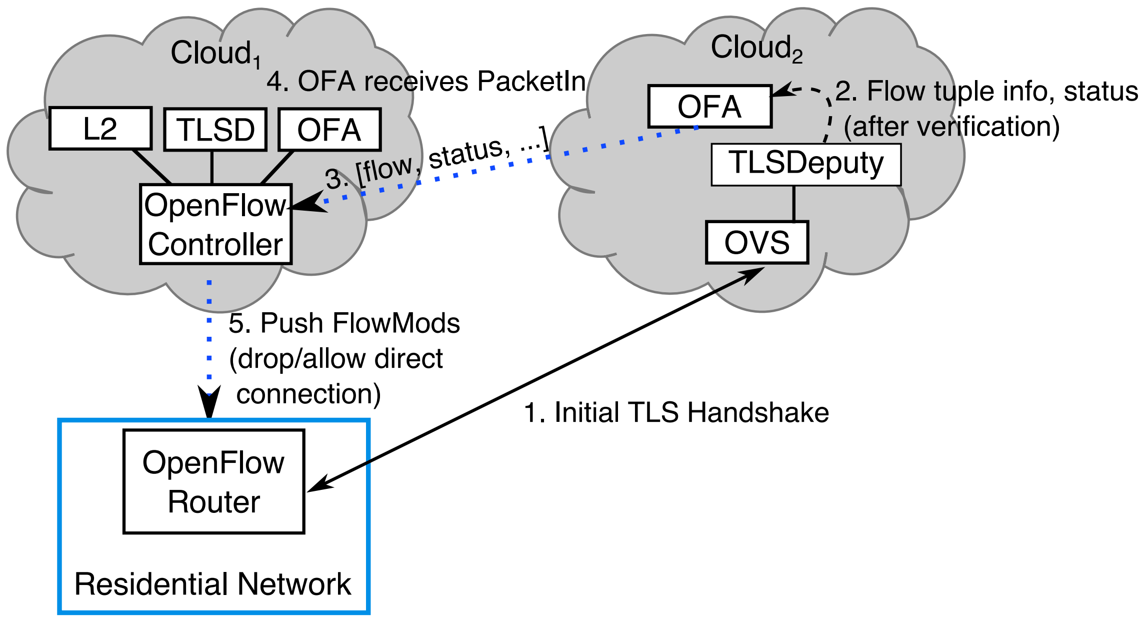


Figure 7.3: Once the TLSDeputy has verified the handshake, it can contact the controller through a custom OpenFlow agent to request the connection be sent directly rather than diverted through the middlebox. The controller can then send FlowMods to the ReSDN router causing packets to be transmitted directly rather than via a tunnel.

as an existing IDS.

7.3.6 Obtaining and Maintaining CRLs

Ideally, our approach maintains an Internet-wide cache of all CRLs. We move towards this goal by initially crawling the top 1 million Alexa domains [6] and obtaining the CRLs for each certificate in a given chain. The initial scan recovered 1,608 potentially reachable CRLs of which 1,495 were retrieved.

Our ideal goal is to maintain a complete list of all CRLs used on the Internet. As a result, anytime TLSDeputy encounters a certificate with a CRL not in the database we add the URI to a list of monitored CRLs and immediately begin retrieving it in the background. However, to avoid performance issues, we do not wait to check the CRL for the chain causing the first retrieval. Instead, we will enforce such revocation checks on the next connection that uses the CRL. For example, Apple's Messages application regularly performs background TLS connections that have several CRLs that were not originally in our database. On the first connection, we will not be able to enforce the CRL, but we will be able to do so on the next connection. As we build our CRL database, we retrieve all CRLs every 12 hours, which is more frequent than the majority of the CRL validity lengths in the certificates we found.

Table 7.1: Evaluation of TLSDeputy on IoT and mobile platforms with a revoked leaf certificate

Device Type	Device		Device Verification	TLSDeputy Verification	Device Revocation	TLSDeputy Revocation
IoT	Foscam		✗	✓	✗	✓
	WeMo		✓	✓	✗	✓
Mobile	iPhone	Safari	✓	✓	✗	✓
		Chrome	✓	✓	✗	✓
		Firefox	✓	✓	✗	✓
	Android	Default	✓	✓	✗	✓
		Chrome	✓	✓	✗	✓
		Firefox	✓	✓	✗	✓
Desktop	Mac OS X	Safari	✓	✓	✗	✓
		Chrome	✓	✓	✗	✓
		Firefox	✓	✓	✗	✓
	Linux	Chrome	✓	✓	✗	✓
		Firefox	✓	✓	✗	✓
	Windows	IE	✓	✓	✓	✓
		Chrome	✓	✓	✓	✓
		Firefox	✓	✓	✗	✓

7.4 Implementation

To implement the TLSDeputy, we use custom router firmware on TP-LINK Archer C7 routers. We installed OpenWrt [45] and added the Open vSwitch [145] package for OpenFlow support. We used the POX [31] controller running on Amazon EC2 micro-instance VMs to manage the router. For tunneling, we used GRE tunnels as supported by OVS. This allowed better systematic tunneling control than our approach in Chapter 5, which required a routing agent to direct flows over a Linux GRE tunnel. When the controller detects a new TLS flow, the TLSD module uses these GRE tunnels for directing the TLS handshake through the TLDeputy middlebox.

Our TLSDeputy is a C++ application leveraging the LibreSSL [22] implementation for certificate verification. We implemented our own certificate stripping and parsing functionality. The CRL retrieval and maintenance code were written as scripts. We ran the TLSDeputy MB and controller in separate EC2 micro-instances.

Our OFA application is a custom OpenFlow 1.0 compliant agent that communicates over an OpenFlow connection to the controller and uses a local TCP socket to receive verification information from the TLSDeputy.

7.4.1 Managing MTU Restrictions

Since we are using built-in tunneling support from OVS, we must account for the overhead in bytes associated with GRE tunneling packets starting from Layer 2. The Maximum Transmission Unit (MTU) between networks is typically 1500 bytes. Without accounting for the GRE overhead, our packets could be dropped by intermediate routers before reaching the tunnel endpoint.

One possibility for addressing this issue is to using IP fragmentation to split the packet and have it reassembled at the MB. IP fragmentation is typically avoided when ever possible due to performance concerns. Instead, we use the MB to set the Maximum Segment Size (MSS) to 1400 bytes in the TCP handshake of both the source and the destination. By reducing the MSS in the SYN/SYN+ACK packets, both end-points of the connection will reduce the payload of packets transmitted and thus avoid fragmentation altogether.

7.5 TLSDeputy Evaluation

We evaluate TLSDeputy's security effectiveness using two IoT devices, smartphone web browsers, and web browsers on traditional laptop/desktop operating systems. We then compare the performance of TLSDeputy against traditional certificate verification and revocation from a residential network.

7.5.1 Experimental Setup

For our security evaluation, we use multiple security testing software packages and our own certificate authority. Many IoT devices are hardcoded to communicate with specific servers or domains. Accordingly, we use `mitmproxy` [76] and SSLsplit [150] to determine if these non-browser applications and devices properly verify TLS certificates and detect forgeries. We monitor network traffic from such devices to determine if the device performed revocation checking via OCSP or CRL retrievals. We created a self-signed root CA and a TLS chain consisting of a single intermediate certificate authority. Using the intermediate CA, we signed a leaf certificate for a publicly accessible web server. Our leaf certificate's revocation status was obtainable only via a CRL. After generating the web server's certificate, we immediately revoked it and updated the CRL accordingly. However, the web server was configured to continue using the revoked certificate.

The two IoT devices we use in testing are a Foscam IP camera, which is used for home surveillance, and a Belkin WeMo power outlet that can be turned on or off through a smartphone application.

7.5.2 Security Effectiveness

Our security evaluation focuses on IoT, mobile devices, and desktop browsers that operate within the home network. We compare how TLSDeputy operates in comparison to the software embedded on two IoT devices, both of which have known security vulnerabilities [23, 42]. We also perform tests using mobile devices using several major browser platforms. The results of security evaluation are shown in Table 7.3.6.

We first describe the IoT device results. Unsurprisingly, neither the Foscam or WeMo performed any type of revocation. WeMo has a reported verification vulnerability that a certificate store is not stored locally on the device. In our testing, we did not find that our device was vulnerable to MITM attacks. However, we did find the Foscam was vulnerable to a MITM attack. Foscam's configuration allows users to setup notifications of motion detection with images through an email.

During configuration, the user must provide a mail server configuration, including a domain name and port, and if authentication is required, a username and password as well. Mail servers such as Gmail require a TLS connection for sending and receiving email. Our research found that the Foscam is indeed vulnerable to a MITM attack on the communication between the camera and the Gmail mail servers, which can expose a user's Gmail username and password. We found none of the listed CVE's for Foscam [23] discuss TLS vulnerabilities and conclude this was previously undocumented. Fortunately, our TLSDeputy system is able to detect and block this MITM attack without requiring software updates to the Foscam or support from the manufacturer[4]. Without TLSDeputy, it would be very difficult to determine if a MITM attack was occurring on any IoT device.

During our evaluation, we expected that mobile browsers would perform proper verification. Indeed, without installing our root certificate on the mobile device, all browsers detected the certificate was untrusted, stopped the connection, and notified the user. After these tests, we installed our root certificate on all each device in order to have the browsers trust the certificate chain and then attempt a new TLS connection. After establishing the connection, none of the mobile browsers we tested performed revocation checks on our server's certificate, which corroborates recent research [124]. In contrast to Liu's work, we found that the newest version of Safari (v9) did not properly check our CRL for revocation. Their tests covered through v8. Additionally, we found that Chrome v49 did properly check the revocation status. Liu *et al.* [124] found that Chrome v44 only performed this check for Extended Validation (EV) certificates, which our certificate was not. Chrome may have recently updated its revocation process. Again, TLSDeputy uses its cached CRL to block connections for each browser as shown in Table 7.3.6, protecting even devices and applications that fail to perform the appropriate verification or revocation checks.

7.5.3 Performance Results

Our performance experiments present two different comparisons. We first consider the end-to-end performance of using TLSDeputy versus traditional end-host verification when only considering the leaf certificate for revocation. Our other performance experiment compares TLSDeputy to full path revocation checks using CRLs. The results were obtained from a residential cable network in Massachusetts with Amazon EC2 instances hosted in the North Virginia data center.

TLS Verification and Revocation Overhead

Virtually no desktop or mobile browser performs full chain verification using CRL or OCSP. Given the frequency of browsers only checking leaf certificates, we perform head-to-head performance measurements over 40,000 random domains using OCSP and CRLs to performing revocation checks on leaf certificates. We then performed connections using TLSDeputy to the same domains.

For OCSP and CRL leaf certificate revocation checking, we first performed a TLS handshake with only verification (i.e., not checking for revocation). Upon verification, we obtained the leaf certificate's OCSP URL and each of the certificates provided during the handshake to check the

[4]Prior to publishing this work, we contacted both manufacturers and disclosed these vulnerabilities and suggested remediation approaches.

leaf certificate's revocation status. We then added the time to perform the OCSP check to the TLS handshake time. Similarly, we obtained the CRL distribution point from the TLS handshake and performed a file retrieval on the CRL. The time taken to retrieve the CRL file was added to the base TLS handshake time. Lastly, we initiated a new TLS handshake with TLSDeputy enabled, but allowed TLSDeputy to also perform revocation checks on intermediate certificates. The results are presented in Figure 7.4 and shows that TLSDeputy adds roughly 0.5 seconds to the median of an TLS handshake.

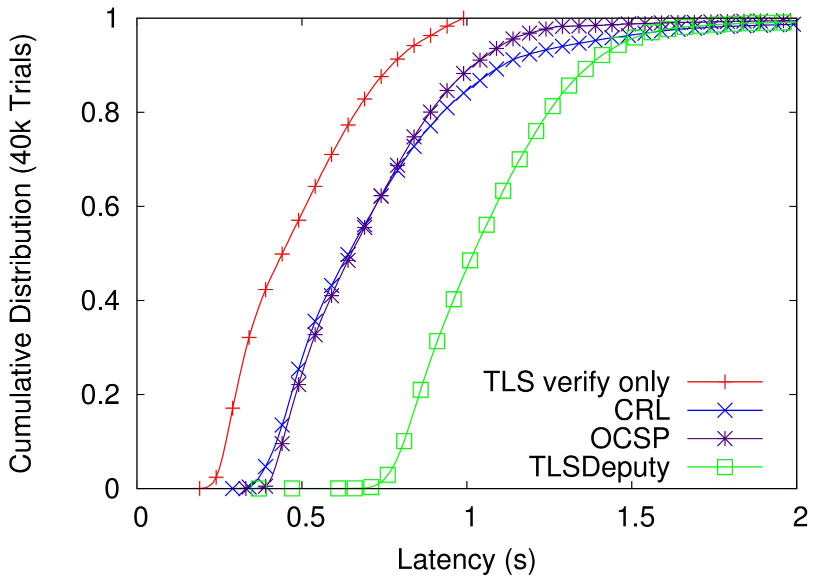


Figure 7.4: Leaf certificate verification comparison between CRL, OCSP, and TLSDeputy over 40k random domains.

Full Chain Revocation Overhead

Only 48.5% of intermediate certificates (which excludes leaf certificate CRLs) offer OCSP for revocation checking [124]. This low number of OCSP responders means that the majority of full path revocation checks require CRLs. To better understand the impacts of full chain revocation checks, we perform an additional experiment using 10,000 random domains which have two or more CRLs in the chain. Similar to our previous experiment, we first initiate a TLS handshake and then retrieve each CRL in the path while accumulating the total time for the connection and each CRL retrieval. The results of this experiment on shown in Figure 7.5 and show the overhead associated with full chain revocation checks using CRLs is comparable to TLSDeputy's performance.

Viability in Practice

In performing the verification across 40,000 domains, we found that TLSDeputy was viable in practice and was able to determine which TLS connections were valid and which were not.

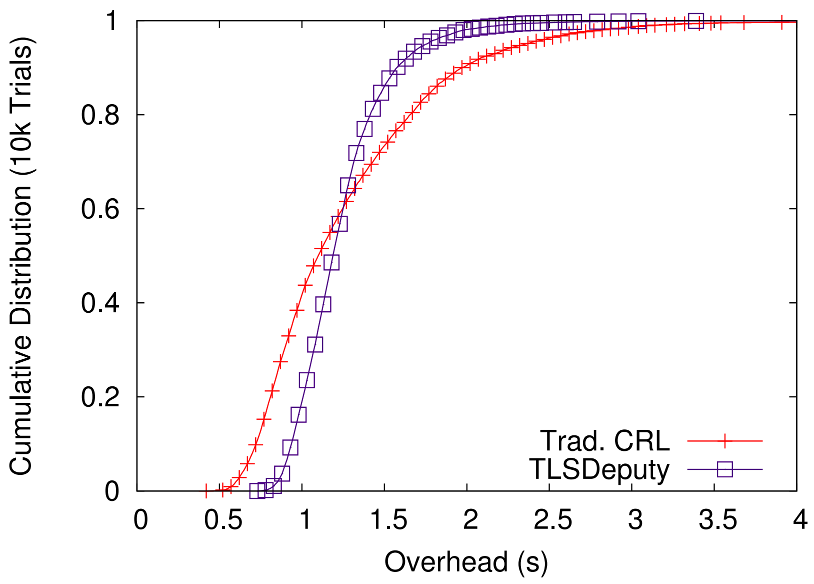


Figure 7.5: Complete chain verification using CRLs.

7.5.4 Evaluation Summary

Our approach is able to protect vulnerable devices, including IoT devices, from connecting to servers with invalid certificates. Further, we are able to protect many IoT, mobile, and desktop devices that do not properly check for certificate revocation. The performance costs for doing so are comparable to a full chain CRL verification at the client. Essentially, our middlebox strategy is able to provide whole network protections for a residential network at roughly the same cost of doing the appropriate verifications at each end device.

7.6 Discussion

While we have focused on TLS in this paper, the same approach is viable for other security protocols such as SSH and IPSec. In particular, SSH's leap-of-faith security approach, in which a user may accept a public key for a server without verifying it, has recognized security risks [58]. We can eliminate the need for a leap-of-faith by combining the use of DNSSEC and the `SSHFP` resource record [152]. Our middlebox could intercept DNS responses, cryptographically verify the `SSHFP` records using DNSSEC, and store the destination IP address and SSH fingerprint for each server in a temporary database. For any SSH connections to known IP addresses, the middlebox would then verify the public key matched. With our tool, an organization could configure DNSSEC and `SSHFP` records to ensure any clients using our approach would be protected from SSH man-in-the-middle attacks during the first SSH connection.

We can protect IPSec authenticity in a manner similar to SSH. Using DNSSEC and the `KEY` resource record [148], the middlebox can perform the appropriate verification to ensure the IPSec server's authenticity.

From a cost perspective, our development and evaluation cost approximately \$20 per month for cloud hosting. The costs included two always-on VMs, network traffic transmission, and disk storage, with the majority of the cost associated with the VM uptime. Given our minimal CPU and memory overheads, multiple residential networks could easily share these VMs. Practically, a third-party security provider could run cloud-based VMs to provide TLSDeputy services to large numbers of residential users and achieve economies of scale.

7.7 Conclusion

In this work, we present TLSDeputy, a system that allows residential networks to ensure they only connect to properly verified TLS servers. We have shown the approach offers valuable security protections for IoT, mobile, and desktop devices and that the performance is comparable to correct client-side verification measures. Finally, using a set of 40,000 servers, we have demonstrated the approach is capable of verifying connections to top TLS destinations and can immediately be deployed to residential networks.

Chapter 8

Evaluating Cloud-Based ReSDN Controllers and Middleboxes

8.1 Introduction

Some researchers have proposed addressing the lack of expertise and restricted hardware limitations in residential networks by installing a new type of hardware device, called virtual customer premises equipment (vCPE) [33], in place of existing residential routers. These vCPE devices are designed to work with support from the user's Internet Service Provider (ISP), which must create and manage the appropriate middleboxes to offer management services and security protections. Unfortunately, the deployment of vCPE solutions are limited and residential users rarely have multiple broadband ISP options in the United States [24], leaving most residential users without access to these services. Instead, our prior approach leverages existing hardware in the home and allows a third-party to immediately deploy new services in the cloud without requiring ISP support.

In Chapters 5, 6, and 7, we proposed and demonstrated the feasibility of our ReSDN infrastructure that adds SDN functionality to existing commodity home routers, via the OpenFlow protocol. With SDN, network operators are able to programmatically control network switches and routers from a centralized controller. That residential SDN approach, shown in Figure 8.1, uses an OpenFlow controller that runs on a public cloud server and manages the traffic flow on a residential network by directing the traffic through the appropriate network function virtualization (NFV) middleboxes.

While our other work has demonstrated the viability of the cloud-based controller and middlebox approach for a single well-connected urban residence, it is unclear how viable such an approach would be across a larger user base. To understand the performance impact across *diverse* residential networks, we conduct two measurement studies. The first study involves 270 geographically distributed residential users to evaluate the latency impacts associated with using cloud-based SDN controllers. The second study uses a different set of 13 geographically distributed users to understand the impact middleboxes have on throughput. Our key contributions are as follows:

- We identify and categorize three classes of cloud-based network security and management modules, (1) controller-centric modules, (2) partial connection middlebox modules, and (3) full-connection middlebox modules, and describe the key characteristics of each.
- We evaluate the latency and bandwidth impacts of cloud-hosted OpenFlow controllers on real-world residential connections. We find that 90% of residential users have acceptable

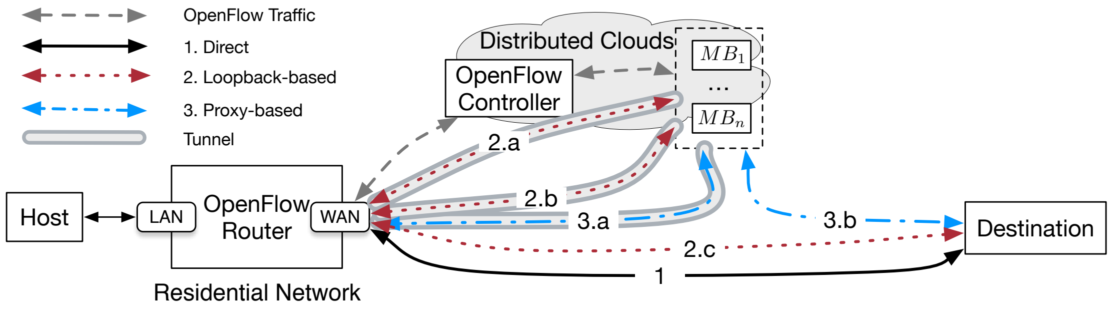


Figure 8.1: System diagram for residential SDN and middlebox solutions. These approaches use a SDN-capable router within the home, a cloud-based OpenFlow controller, and a number of cloud-hosted middleboxes for NFV support. Paths labeled as 2.a, 2.b and 3.a use pre-established tunnels.

performance with at least one public cloud location for a potential OpenFlow controller within a 50 millisecond round trip time (RTT). This result also shows the need for more public cloud data center locations to accommodate the remaining 10% of residential users.

- We investigate the impact of the latency inherent in cloud-based controllers on user-perceived performance. We focus on latency-sensitive web browsing traffic that dominates residential traffic. We measure the page loading time (PLT) associated with the top 100 Alexa websites and explore different factors that contribute to the PLT differences. Our results show negligible performance degradation for 80% of Alexa sites even with an OpenFlow controller with a 50 ms RTT. In other cases, the performance impact is similar to that of browser-based advertisement blocking extensions, which are used by millions of Internet users.

- We evaluate the impact of cloud-based middleboxes on residential connections while focusing on full-connection modules. We find that upload bandwidth is largely unaffected by cloud-based middleboxes. The impact on download bandwidth is more nuanced and can be affected by bandwidth shaping policies at the cloud provider.

8.2 Classes of SDN and Middlebox Modules

Before we can analyze the performance impacts of cloud-based OpenFlow controllers and middleboxes on residential networks, we must first identify how these systems are effectively used in practice. We have identified three broad classes of modules that may run on the controller. These modules characterize how SDNs and middleboxes can be used to manage and protect residential networks. They are as follows:

- **Class 1: Controller-centric Modules:** For these modules, the controller only needs to see the initial packet in one or both directions of a new network connection. Such modules include stateless firewalls, DNS blacklists, and connection loggers. These modules can be implemented at an SDN controller without requiring a separate middlebox that would analyze subsequent packets in the flow. The network traffic for these modules are depicted using the dashed OpenFlow traffic and the solid direct traffic lines in Figure 8.1.

- **Class 2: Partial Connection Middlebox Modules:** These modules must consume a relatively small portion of a connection's actual payload to function correctly, but do not need to be involved in the full connection. Such modules include deep-packet traffic classification tools or security tools that validate the initial handshaking process of a connection, such as our own TLSDeputy [167] module. These modules can use a "loopback" approach, as shown in lines 2.a and 2.b in Figure 8.1, in which the OpenFlow switch essentially *temporarily* redirects communication for the connection through a tunnel to a middlebox before receiving it again and delivering it to the destination, as shown in line 2.c. The tunneling allows the middlebox to inspect the payload. Once the middlebox has finished vetting the connection, it then informs the SDN controller to remove the indirect looping process. Accordingly, the remainder of the connection proceeds directly between the end-hosts without middlebox involvement, as shown by line 2.c in Figure 8.1.

- **Class 3: Full Connection Middlebox Modules:** This class of modules require that all packets in the flow, in both directions, be inspected by the middlebox module for the life of the connection. Example modules include intrusion detection systems (IDSes) and anonymizing proxies, since any packets that bypass the middlebox would undermine the module's mission. The traffic pattern for this approach is shown with lines 3.a and 3.b in Figure 8.1.

8.2.1 Performance Implications for Module Classes

With this basic classification of modules, we can begin to discuss the key performance characteristics of each. For Class 1 and Class 2 modules, bandwidth, jitter, and packet loss rates are less important than they are for Class 3 modules. This is because Class 1 and Class 2 modules are typically only associated with a flow during the initial connection and application-layer handshaking process, which is less sensitive to these characteristics than payload-centric portion of the flow.

However, network latency between residential routers and cloud-based OpenFlow controllers can be important for all three classes. For Class 3 modules, which impact the connection's payload packets, latency-sensitive applications such as online games [75] will require that the diversion through a middlebox does not substantially affect the RTT. For Class 1 and Class 2 modules, the importance of the latency for the initial packets may vary. For long-lived, high volume flows, often referred to as *elephant flows*, the initial latency of the round-trip to the controller is less important since it can be amortized across the length of the flow. However, other flows, commonly called *mice flows*, can be short and low volume. In these cases, the latency of the initial connection can have a high impact, especially if there are numerous mice flows associated with a single user interaction. In particular, web browsing can be the epitome of mice flows since accessing a single web page may establish many separate connections. Further, for web browsing, the separate connections may have dependencies. As an example, loading an HTML document may alert the browser that it must load a JavaScript file, which causes the browser to then load an image. If the latency to the controller is high, it will have cumulative effects for each new iteration through a web page's dependency chain. As a result, web browsing can represent the worst case scenario for OpenFlow when each flow must be elevated to the controller.

8.2.2 Measurement Objectives

Since Class 1 and Class 2 modules have largely similar networking requirements, we consider them to be manifestations of the same research question: Where should we place an OpenFlow controller to minimize its latency impacts on connections? For Class 2 modules, we could place the middlebox in the same hosting location as the controller. We discuss this problem in detail in Section 8.4.

For Class 3 modules, throughput is a key requirement for middlebox placement. We thus consider the problem of appropriately positioning payload-consuming middleboxes in Section 8.5.

However, before analyzing the controller and middlebox placement problems, we first describe our measurements and data collection efforts for the residential networks we measured.

8.3 Measurement Methodology

In this section, we detail our methodology for measuring and understanding the feasibility of outsourcing an OpenFlow controller to a cloud server given the current *residential network* connectivity present in the continental United States (US). We focus on the continental US given its broad geographic region, diverse last-mile network connectivity, and its mixture of urban and rural residences. We leverage four popular public cloud platforms: Amazon EC2, Google Cloud Platform, Microsoft Azure, and Digital Ocean. Using these services, we host a total of 12 measurement servers inside virtual machines (VMs). In addition, we also include a server running in a VM at our university. These servers are geographically spread across US, as shown in Figure 8.2(a), and are used in either a proxy or measurement server role. We then recruit residential users to perform two separate sets of measurements using the above infrastructure. We collect network-level data for both network latency and bandwidth to determine the performance implications of a cloud-based OpenFlow controller on residential networks.

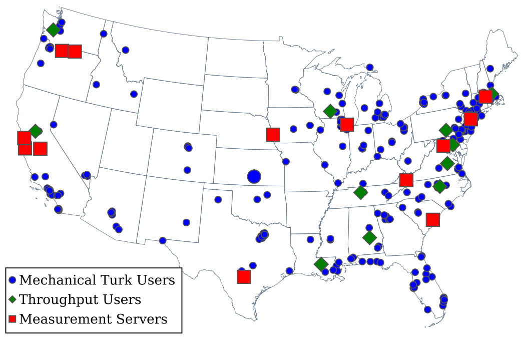
We now describe our measurements and participant recruiting methodology in greater detail.

8.3.1 Cloud Controller Latency Measurement

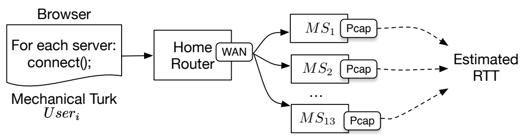
As discussed in Section 8.2, the network latency between the SDN controller and the residential router is an important factor that could influence residential network performance. To gauge the feasibility of our ReSDN architecture given the current public cloud infrastructure, we conducted a two-week measurement in August 2016.

Using Amazon's Mechanical Turk service [140] we recruited participants and provided them with modest compensation to visit a speed testing website [1] that we hosted at our institution. Through that service, we initially recruited a total of 497 unique participants. However, we had to exclude users that did not meet our eligibility criteria, namely that the user is located in the United

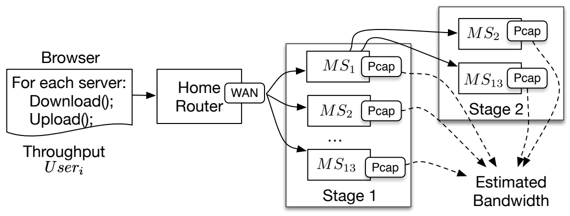
[1] Available at `http://speedtest.wpi.edu/`.



(a) Mechanical Turk users and cloud infrastructure



(b) RTT measurement



(c) Throughput Measurement

Figure 8.2: Illustration of our data collection and measurement methodology. Figure 8.2(a) shows the distribution of Mechanical Turk users, throughput users, and measurement servers[3]. Figure 8.2(b) shows the process an Mechanical Turk client performs allowing us to determine RTT. Stage 1 in Figure 8.2(c) shows the throughput clients perform a series of downloads and uploads directly to all servers. Using JavaScript approximations in Stage 2, these clients perform an addition download and upload using the fastest server as a proxy to the slowest and next fastest server.

States and is using a residential network connection (which excludes VPNs, cellular connections, and corporate networks). We used a combination of reverse DNS, IP geolocation databases, and an examination of the IP address's associated network provider, we filtered our participants to a total of 270 eligible participants.

During the speed test, the residential user's browser first downloads a JavaScript file that contains URLs that can be used to access our distributed cloud VMs. The browser then runs the script to establish HTTP connections to all our VM servers. We calculate the round trip time (RTT) between the residential user and all cloud servers using packet captures collected at the VM servers. We present and analyze these results in Section 8.4.

8.3.2 Cloud Middlebox Throughput Measurement

We perform a throughput measurement to determine what impact, if any, a cloud-hosted middlebox would have on the end-to-end throughput between a residential user and a cloud server. In theory, the impact should be relatively small if the cloud-hosted middlebox has good network connectivity since the "last mile" connection between the residential user and the user's Internet Service Provider (ISP) is often the throughput bottleneck and cloud providers often have high available bandwidth and have a successful track record of hosting servers that demand high throughput, like video streaming providers. However, in practice, rerouting network traffic can expose new throughput bottlenecks or congestion on either the original or rerouted path. Accordingly, we empirically examine the impact of cloud middleboxes.

We recruited 13 users, comprised of colleagues, family, and friends, from 11 different states

[2]We geolocated the participants using the MaxMind database [14].

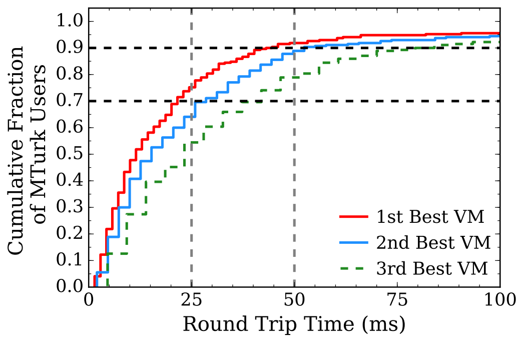
across the US, shown in Figure 8.2(a). Again, we use the same measurement infrastructure[3] and the JavaScript-based approach to perform throughput measurements. To measure the end-to-end throughput between a residential user and a cloud server, the user's browser first downloads an 18 MB file from the server and then uploads a 10 MB file to the same server. The browser calculates both the download and upload throughput by dividing the file size by the transmission time as measured in JavaScript. For both upload and download throughput, we separately sort all cloud servers in descending order based on throughput measurements and labeled as $\{MS_1, MS_2, \ldots, MS_{13}\}$, where MS_1 represents the server with the best throughput between itself and the client and MS_{13} represents the server with the worst throughput between itself and the client. We then perform two more throughput measurements: 1) an end-to-end throughput measurement from the client to M_1 to M_2 and 2) a measurement from the client to M_1 to M_{13}. Intuitively, these extra two measurements show us what throughput impact one would expect for the client if a cloud-based middlebox was hosted at the M_1 site and provided connectivity to the other sites. The first case is most likely to manifest a throughput degradation since the throughput to M_2 is relatively good. The second case, via M_{13}, has relatively poor throughput and would be least likely to have a throughput degradation. Accordingly, these are useful sample points for bounding the likely throughput a client would see when tunneling via a well-positioned cloud middlebox in practice. The corresponding bandwidth results and the performance implications of using cloud middleboxes are presented and analyzed in Section 8.5.

8.4 The Cloud Controller Placement Problem

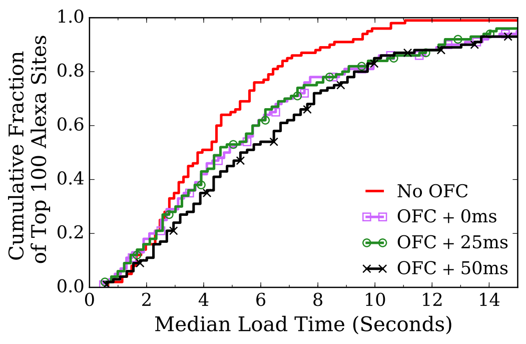
The placement of an OpenFlow controller with respect to its controlled switches has previously been recognized as an important problem [106]. Latency is the primary consideration when placing the controller. In enterprise or data center networks, the controller can be placed in the same LAN. Unfortunately, such in-network placement is often infeasible for residential network settings. When studying the controller placement problem, we are interested in understanding the feasibility of deploying a cloud-based controller for residential users.

Figure 8.3(a) shows the network latency performance for US residential users with the current cloud infrastructure. We find that more than 70% of Mechanical Turk users are within a 25 ms RTT and roughly 90% are within a 50 ms RTT to more than one cloud server we control. Our latency measurement results indicate the promise of hosting an OpenFlow controller within a reasonable RTT to a large fraction of US-based users. To further understand the impact of deploying a cloud-based controller on residential users' end-to-end experience, we design a series of experiments that evaluate the web browsing performance based on our latency data, as shown in Figure 8.4.1.

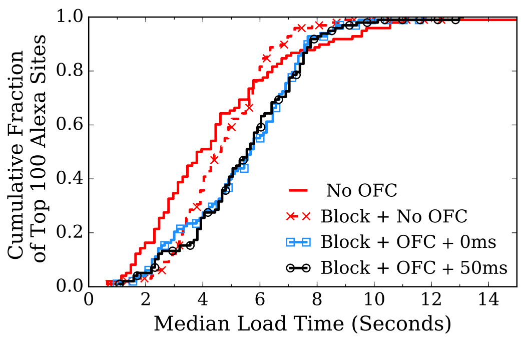
[3]To avoid skewing throughput results, we excluded our university server from this measurement study due to its high bandwidth allowance.



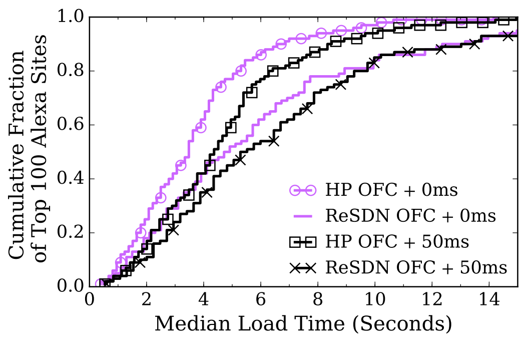
(a) Distribution of RTT between Mechanical Turk users and cloud servers



(b) Page Loading Time with ReSDN



(c) ReSDN PLT Blocking Non-essential Connections



(d) ReSDN vs. HP Switch

Figure 8.3: RTT measurements and the resulting page loading time (PLT) analysis based on those RTT measurements. We compare the PLTs by fetching top Alexa 100 websites at different controller latencies on our ReSDN switch and see how PLT is affected by blocking non-essential connections. Finally, we compare our consumer-grade switch to an enterprise-grade HP switch.

8.4.1 Quantifying Performance Impact with Page Load Time

In this experiment, we quantify and analyze the performance impact of our residential SDN architecture on residential users by measuring the page load time (PLT) of popular domains from a residential network. As noted before, web-related traffic is often composed of mice flows, which are the worst-case scenario for controller latency since the latency cannot be amortized over the length of a longer connection. Further, web traffic is an important category that dominates residential traffic [127]. Accordingly, PLT is a useful indicator for estimating the most severe impact on a user's network traffic since it captures both mice flow behavior and dependencies between connections.

When a residential user visits a domain using our residential SDN architecture, each connection to fetch remote resources, from website servers, CDN servers or advertisement networks, needs to be approved by the Floodlight SDN controller [12]. By controlling each flow, we examine fine-grained flow control, which enables better situational awareness and security applications [96]. As such, our PLT measurements require every new flow to be independently approved by the Floodlight OpenFlow controller and results in an additional entry in the flow table.

The PLT is defined as the time interval between the start of the first connection and the end of the last connection, which we capture using events triggered in the Chrome browser as discussed in [64]. Intuitively, assuming the residential user's request to a domain is fulfilled by the same set of end resources, PLT can be impacted by (1) the network latency between residential SDN router and the OpenFlow controller, (2) the maximum length of dependent network connections, and (3) the residential router's processing power.

We explore three variables that can impact PLT, although only the network latency can be easily optimized directly in a residential SDN architecture. To measure the PLT of a particular domain, we modified a popular open source Chrome extension [16] to record the time it takes for the page load event to occur [64, 175]. We repeat the PLT measurement 25 times for each site, each time with a clean browser cache. We then report the median PLT for that site. We perform the same measurement for the top 100 Alexa US domains.

To study the impact of RTTs on our residential SDN architecture, we vary the network latency from 0 ms to 50 ms at the controller's output interface based on our Mechanical Turk-based residential network measurement. Based on our results (shown in Figure 8.3(a)), 90% of users can reach a cloud-based controller within a 50 ms RTT. Therefore, we believe the performance degradation observed with a 50 ms RTT is a reasonable upper bound for our subsequent trials. In Figure 8.3(b), we plot the median PLT for all Top 100 Alexa domains. It shows that redirecting all new connections to an OpenFlow controller increases the PLT, from 6 seconds to 10 seconds, at the 80th percentile of Alexa sites. As the RTTs increase to 50 ms, the median PLT has a modest increase at the 80th percentile of Alexa sites. In all, we conclude that the performance degradation in the form of median PLTs is mostly attributed to the existence of the controller and the flow elevation process.

8.4.2 Impact of Advertisements on Page Load Time

We next study the impact of the length of dependent network connections on the PLT. In our connections, we see that the loading supplementary background content, such as advertisements and analytics scripts, usually happen recursively. In addition, such background content is not essential to website functionality and is specifically blocked by many users [174]. Based on these observations, we repeat the PLT experiment by blocking these non-essential network connections with two popular blocking extensions, AdBlock Pro [4] and Ghostery [15], and analyzing the impact on connections without OpenFlow with those using OpenFlow.

Figure 8.3(c) shows that running these browser extensions degrades median PLTs for up to 2 seconds. In an extreme example, the website for a home improvement store, the use of these blocking extensions increased the median PLT by 3.38 seconds. However, prior work has shown that such a degradation in PLT is acceptable for end users of such blocking tools [174]. Intuitively, these extensions block multiple network connections and thus avoid elevating some flow decisions to the controller. Accordingly, the median PLTs are improved significantly when compared to elevating all network connections to the controller. We manually inspect the top five domains, shown in Table 8.1, that benefit the most from blocking non-essential network connections. These results suggest a correlation between median PLT decrease and the number of blocked connections, which matches the simple intuition that a greater reduction in connections yields a greater reduction in PLT.

While OpenFlow may have substantial impact on residential network PLTs in the worst case, it seems the most affected content is actually related to advertisements or analytics. Since millions of users actively try to block such content from ever loading, the remaining users may tolerate delays in obtaining and displaying such content.

8.4.3 Impact of Router Hardware on Page Load Time

We also investigated whether the hardware of consumer-grade routers would be a factor for this approach. We compared an enterprise-grade HP 2920-24G OpenFlow switch with a consumer-grade TP-Link Archer C7 router running Open vSwitch. As one might expect, the enterprise switch nearly always outperforms the consumer router in the 0 ms and 50 ms latency environments, as shown in Figure 8.3(d). The HP switch has multiple advantages, including more memory and hardware flow tables. However, our measurement process also gives the HP switch a built-in advantage: the TP-Link router supports TLS and was enabled in our experiments, since a practical deployment requires TLS for the OpenFlow connection. Since the HP switch does not support TLS connections, it was not responsible for the inherent encryption and decryption operations for each message. The overheads associated with TLS are unclear without further experimentation.

The advantages of the HP switch were insignificant for some sites but very large for other sites. Our other results used the consumer-grade router and thus provide a conservative estimate of the performance available to residential users. As consumer router hardware improves, we may see additional improvements in PLTs.

8.4.4 Controller Latency Summary

Our measurement results indicate there is a 3.5 second increase in median PLTs in the worse case scenario when using a cloud-hosted controller. As the cloud becomes more distributed, we expect the median PLTs to drop proportionally to the minimal cloud latency. Residential users experience similar PLTs when running popular browser plugins to block unessential connections compared to running OpenFlow controller with 50 ms network latency. This leads us to believe our results will be broadly acceptable to, at least, a growing number of users that deploy ad blockers, which has reached 50 million in US and 236 million worldwide [2]. Further, as consumer router hardware improves, the overall latency may also improve.

8.5 Cloud Middlebox Placement

The middlebox placement problem, also called the NFV placement problem [62], has received less attention from the research community than the controller placement problem. However, middlebox placement is important for residential users and we now explore the additional requirements for middleboxes and their impact on selecting middlebox VM data center locations.

Table 8.1: These five sites benefited the most by blocking non-essential connections. The PLT reduction is calculated as the difference of median load times by turning off/on blocking when using a 50ms RTT to the controller.

Site	Median PLT Reduction (Seconds)	Num. of Blocked Connections
`huffingtonpost.com`	15.32	50
`drudgereport.com`	14.59	12
`businessinsider.com`	12.64	41
`dailymail.co.uk`	11.16	57
`cnet.com`	10.49	34

Table 8.2: Residential bandwidth measurement results for 13 users and 12 cloud servers. We compare the throughput impact of using the best throughput server as a proxy to the worst throughput server and to the second best throughput server. The percentage and throughput impact are a comparison from the proxied connection to the direct connection.

Percentage Differences	Throughput from Best to Worst	Throughput from Best to 2nd Best	Absolute Differences	Throughput from Best to Worst	Throughput from Best to 2nd Best
Improved Download	8	3	Improved Download	8	3
$0\% \leq x \leq 25\%$	1	0	0 Mbps $\leq x \leq 2$ Mbps	1	0
$25\% < x \leq 50\%$	2	0	2 Mbps $< x \leq 5$ Mbps	4	2
$50\% < x \leq 75\%$	1	1	5 Mbps $< x \leq 10$ Mbps	2	0
$75\% < x \leq 100\%$	0	0	10 Mbps $< x \leq 20$ Mbps	0	0
$100\% < x$	4	2	20 Mbps $< x$	1	1
Improved Upload	12	5	Improved Upload	12	5
$0\% \leq x \leq 25\%$	4	4	0 Mbps $\leq x \leq 2$ Mbps	6	5
$25\% < x \leq 50\%$	2	1	2 Mbps $< x \leq 5$ Mbps	4	0
$50\% < x \leq 75\%$	2	0	5 Mbps $< x \leq 10$ Mbps	1	0
$75\% < x \leq 100\%$	0	0	10 Mbps $< x \leq 20$ Mbps	0	0
$100\% < x$	4	0	20 Mbps $< x$	1	0
Degraded Download	5	10	Degraded Download	5	10
$0\% < x \leq 25\%$	3	4	0 Mbps $< x \leq 2$ Mbps	1	3
$25\% < x \leq 50\%$	2	3	2 Mbps $< x \leq 5$ Mbps	4	3
$50\% < x \leq 75\%$	0	1	5 Mbps $< x \leq 10$ Mbps	0	2
$75\% < x \leq 100\%$	0	2	10 Mbps $< x \leq 20$ Mbps	0	2
$100\% < x$	0	0	20 Mbps $< x$	0	0
Degraded Upload	1	8	Degraded Upload	1	8
$0\% < x \leq 25\%$	0	2	0 Mbps $< x \leq 2$ Mbps	0	2
$25\% < x \leq 50\%$	1	0	2 Mbps $< x \leq 5$ Mbps	1	3
$50\% < x \leq 75\%$	0	3	5 Mbps $< x \leq 10$ Mbps	0	2
$75\% < x \leq 100\%$	0	3	10 Mbps $< x \leq 20$ Mbps	0	1
$100\% < x$	0	0	20 Mbps $< x$	0	0

In this section we focus on Class 3 (full connection middlebox) modules with the goal of maximizing the connection throughput. We conducted our throughput experiments as described in Section 8.3.2 and present the results in Table 8.2. Using packet captures at each server, we calculate the throughput for both direct connections and through connections that proxy through a middlebox. We then calculate the percentage of throughput improvement or degradation associated with the proxy. Importantly, the server that provides the best download throughput may not be the same as the server that provides the best upload throughput.

In the second and fifth columns of Table 8.2, we show the percentage change and absolute change in throughput when traffic is proxied using the client's highest throughput server when communicating with the lowest throughput server. These results show that when using the best throughput server as a proxy to the worst, the majority of residential users either see an improvement in throughput or no change in both directions. This experiment highlights scenarios where we may not only to prevent degrading throughput, but may even improve it. In fact, we were able to improve download throughput for 8 users in this scenario and improve upload for 12 users with several increasing over 100%. While a minority of clients experience throughput degradation in the scenario, those degradations are typically small, both in percentage and in terms of raw throughput.

In the third and sixth columns of Table 8.2, we see the impact on throughput when the best throughput server is used as a proxy to reach the second best throughput server. This experiment shows what happens when we proxy to servers that already provide the user with high throughput. As expected, proxying from the best to the second best proxy frequently hurts performance. This is intuitive: there is little margin for improvement, but the path is longer and thus more susceptible to traversing congested peering points. In some cases, the degradation is modest, but in others, it can be substantial.

This analysis demonstrates that having multiple options for hosting cloud middleboxes can have a significant impact. Further, it highlights that the optimal middlebox may depend on the destination: a well chosen middlebox can actually improve throughput but even a normally good middlebox can degrade throughput for some destinations. Accordingly, cloud middlebox providers may need to maintain a well-constructed cloud measurement infrastructure to dynamically determine the best middleboxes based both on the client and its intended destination on a per connection basis.

8.6 Discussion

When considering the deployment of cloud-based OpenFlow controllers or middleboxes for residential networks, there are a couple points worthy of further consideration: 1) the impact of various OpenFlow elevation models and 2) the accuracy of client provided measurements. We now briefly discuss these points.

8.6.1 Controller Elevation Models

OpenFlow controllers can be configured with different types of policy and rule creation strategies which directly affect the number of elevations required for a new connection. One type of policy, using coarse-grain rules, employs wildcards for network addresses or transport layer ports. With such wildcarding, it is possible for multiple network connections to use the same policy rule. This allows the OpenFlow switch, such as a residential router, to manage subsequent connection matching the policy rule without an additional elevation. In our analysis, we focused on fine-grained flow control policies, in which no wildcards are used for network addresses or ports. This means each new connection requires an elevation to the controller. As a result, our analysis essentially captures the most conservative estimate of the impact of a cloud-hosted OpenFlow controller. The performance of coarse-grain rules would essentially blend our analysis with that of direct connections, with cache hits incurring no additional latency and cache misses having latency similar to our observations.

OpenFlow controllers may choose how they install rules in the OpenFlow switch. Upon receiving an elevation for the first packet in a new flow, the OpenFlow controller may choose to proactively approve the flow bi-directionally or only uni-directionally (essentially, just approving the connection initiator to reach the responder). If the controller installs only a uni-directional rule, any response would also be elevated to the controller, incurring a second round of latency to the controller. In our experiments, our controller installed rules uni-directionally and thus required two elevations to the controller for each new connection (e.g., both the `SYN` packet and for the `SYN+ACK` packets in the TCP handshake). Accordingly, our results are again conservative. A controller installing bidirectional rules would essentially incur half the number of elevations and could be twice as far away while obtaining reasonable performance. Such controllers may be usable for over 90% of residential users since they are less latency sensitive.

8.6.2 Client Measurement Accuracy

The "last mile" connectivity between the residential user and the user's ISP is often the throughput bottleneck. When this is the case, the choice of a cloud-based middlebox should have relatively little impact on throughput. However, in cases where this is not true, such as a congested peering point, the selection of a middlebox may be significantly more complicated. This approach argues for using client-side measurements to help determine the best throughput performance when proxying via different cloud middleboxes. Unfortunately, client throughput measurements may not be reliable in some cases.

In our study, we used JavaScript in web browsers to help us determine the cloud server that offered the highest bandwidth. Most clients had servers that were very close in bandwidth, making measurement precision important for selecting the best server. When we compared the real-time selection from the client's JavaScript with our post analysis via the packet captures at our VMs, we noticed that clients occasionally selected a slightly suboptimal server as their "best" server. This had ramifications for proxying, since the bottleneck between the client and the suboptimal server ensured that the results with other servers, such as the actual optimal server, would necessarily degrade. In production, such measurements would need to be done in a high precision way, such as with low-level packet analysis, to ensure an optimal selection.

During our experiments, we also did not allow any two servers to reside on the same physical cloud location. In a large-scale deployment, users may avoid degradation in some scenarios in which a middlebox server is deployed in the same cloud data center as the destination server. We expect this co-location deployment to be feasible for a large number of destinations in the near term given the continuing cloud outsourcing employed by enterprises [1].

8.7 Conclusion

In this work, we characterize residential network connections to cloud infrastructure. Using Amazon's Mechanical Turk, we recruit 270 participants across the United States and use in-browser instrumentation to direct participants to connect to various cloud instances hosted by 4 major providers in different geographical location. We characterize the connections using JavaScript measurements reported by the client and packet captures on the servers we controlled. With this data, we examine the OpenFlow controller placement problem for residential SDNs and found that 90% of users were within 50 ms of a cloud instance. While this latency is most likely to affect the web browsing experience, due to its interdependent objects and connection characteristics, our subsequent analysis shows this impact primarily slows only advertising and analytics connections. We then examine how best to place middleboxes in cloud environments and find well placed middleboxes have the potential to improve end-to-end connections. With these results, we conclude that residential SDN and middleboxes are feasible for roughly 90% of US users even when limited to publicly available cloud VMs.

Chapter 9

Towards a ReSDN Testbed

9.1 Introduction

In Chapter 5, we introduced our ReSDN infrastructure and in Chapters 6 and 7 showed novel applications of our approach. However, that work has been limited to small scale service deployments, often consisting of a single participant. We explore our vision of a large scale deployment by incrementally deploying our ReSDN infrastructure to participants using an IRB-approved study. By deploying a larger testbed, we can address the following limitations of a single ReSDN deployment:

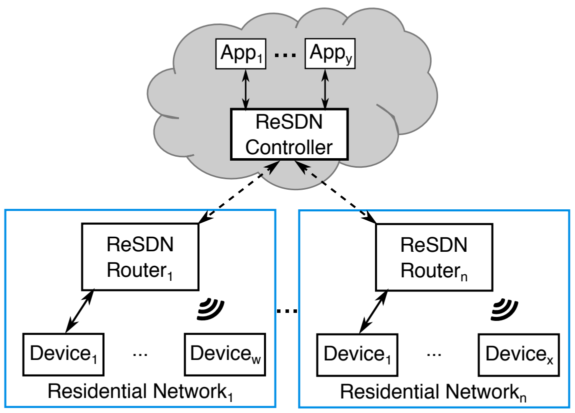
- **Homogeneous workloads**: A single deployment provides homogenous workloads. Typically, our application testing, experimentation, and verification drives the network traffic being generated. While targeted traffic generation simplifies testing and debugging, it may also subtly influence experimental results. Since workloads are only being generated to verify a particular application, the tests are not longitudinal. Accordingly, the long term effects are not well-studied. Having a larger testbed allows us to deploy and test a wider array of applications with traffic generated naturally by participants.

- **Homogeneous devices**: A single deployment testbed reduces the number of devices our controller infrastructure sees. In 2015, the average number of devices on residential networks grew to 5.7 [19]. Device heterogeneity leads to real-world security concerns, particularly with respect to IoT devices [61]. We will miss or be unable to address these concerns with a small testbed. A larger testbed allows us to leverage device heterogeneity for discovering, building, and testing new security solutions.

- **Unrealistic privacy expectations**: The flow-based middlebox infrastructure that we built in the cloud has assumed a holistic view into the residential network's packets. Under this assumption, we were able to freely build applications that consumed the payload of packets such as DNS. In an actual deployment where a third-party is responsible for the control and management of the network, users may not be immediately willing to allow this. Indeed, Feamster's [91] position paper notes that privacy is a major concern that needs to be addressed in outsourcing network security.

9.2 Creating Residential Testbed

We have begun deploying a ReSDN testbed with participants. With IRB approval, we have asked users to participate in our research. Participation requires participants to sign an informed consent document and upon agreeing, users replace their existing home routers with our ReSDN router. As an incentive to participate, users may keep the router after the study completes. Figure 9.1 shows how our testbed allows us to remotely manage each participant's network individually from a centralized controller.



App_1 ··· App_y
ReSDN Controller
ReSDN Router_1
ReSDN Router_n
Device_1 ··· Device_w
Device_1 ··· Device_x
Residential Network_1
Residential Network_n

Figure 9.1: A centralized cloud controller allows us to manage multiple independent home networks.

9.3 Deployment Considerations

There are several practical considerations that need to be addressed before deploying our testbed longer term. These considerations include the placement of the cloud controller, security, and privacy.

9.3.1 Controller Placement

We chose the Floodlight [12] controller for our testbed. Before deploying the router, we must choose where to run our OpenFlow controller. In Chapter 8, we discuss the importance of controller placement and the performance implications associated with cloud controllers. For our testbed, we deploy a single controller in Microsoft's Azure cloud infrastructure located in the East data center. For our initial testbed, this location provides a latency of less than 40 ms to all participants. With a sufficiently large testbed, we would require more geographically distributed controllers.

9.3.2 Securing OpenFlow Communication

ReSDN routers communicate to the cloud controller using the OpenFlow protocol. By default, this communication is unencrypted. As a result, encapsulated packet payloads, such as DNS packets, will also be unencrypted allowing any device between the home router and the cloud controller to

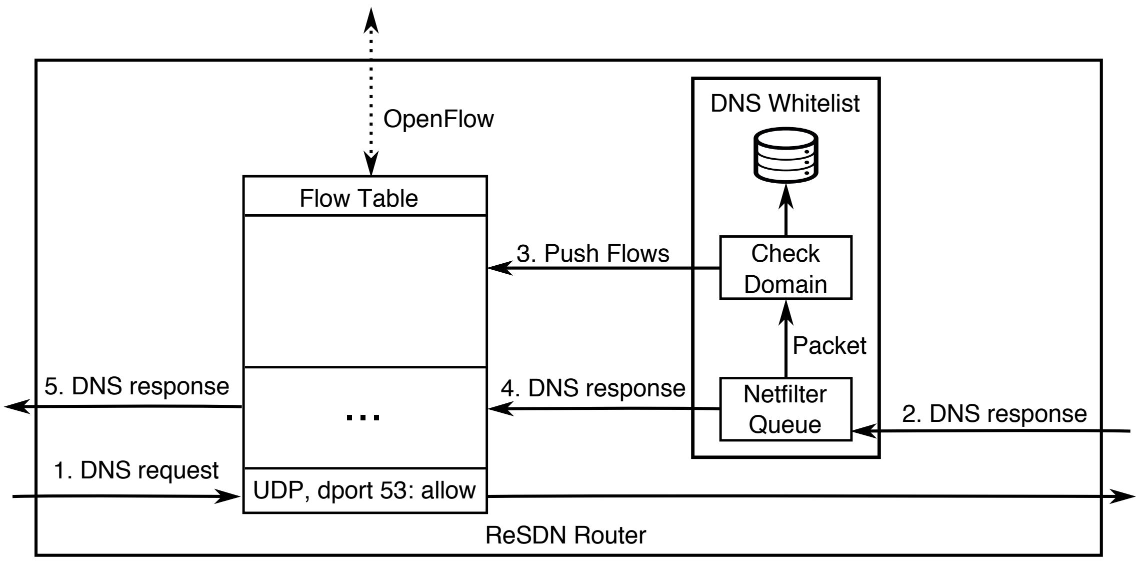


Figure 9.2: ReSDN DNS whitelist processing for better privacy control. A local process parses DNS packets and manually inserts OpenFlow rules as necessary without involving the cloud controller.

view the contents in plaintext. However, the OpenFlow protocol supports using TLS for encrypting communication.

Our ReSDN deployment uses TLS to secure OpenFlow traffic by generating per-switch TLS certificates. The switch has its own certificate embedded in addition to the controller's certificate. Accordingly, we support traditional TLS server and client verification.

9.3.3 Privacy

As per our IRB protocol, we empower participants with the ability to limit the exposure of their network traffic to our ReSDN infrastructure. There are four different ways users are able to influence what traffic we handle.

DNS Whitelisting

Users may explicitly add domain names to a database of domain names. A local process leverages this database to make decisions on what DNS requests the controller sees in addition to any subsequent TCP or UDP connections. Figure 9.2 provides an overview of how DNS whitelisting is implemented. We include an initial database of 10,290 domains across various categories of sites include financial and adult websites. The database is maintained on the router and is not monitored by our infrastructure. Accordingly, management of the database requires the user to navigate to a web page hosted on the router itself.

Deployment Date	Location	Flows Approved	OpenFlow Data (GBs)	# Devices
2017-02-26	MA	972,212	1.56	8
2017-03-04	TN	1,279,053	1.84	8
2017-03-27	MA	2,008,219	2.76	18

Table 9.1: Overview of ReSDN testbed data.

MAC Address Whitelisting

The next option participants have available is to exclude devices based on MAC addresses. MAC addresses can be added by authenticating with the ReSDN overview website[1] and submitting the MAC address. Adding new MACs causes the ReSDN router to restart the connection to the controller. Upon reconnection, the controller pushes static rules for all address resolution protocol (ARP), UDP, and TCP traffic for each whitelisted MAC address.

Blackout Mode

Additionally, participants have available to them the ability to temporarily disable all ReSDN services by entering a "blackout mode" located on the same page as the DNS whitelist functionality. Blackout mode adds entries to the OpenFlow table that causes all traffic to be handled locally without consulting the controller. The entries last for 30 minutes at which point new network flows are sent to the controller for approval again.

Data Removal

As a final option, users have the ability to retroactively remove any data collected by the ReSDN infrastructure. On the ReSDN overview page, participants can choose any length of time of which to remove data. Data removed includes, but is not limited to, the flow level information reported to the OpenFlow controller.

9.4 Testbed Deployment

We have deployed our ReSDN router in three residential cable networks across two states. The deployment has been active for over one month.

9.4.1 ReSDN Data

A goal of a ReSDN testbed was to diversify network connectivity with different workflows and devices. In Table 9.1, we show that we have seen 34 different devices performing network activity. From these 34 devices, we have approved over 1 million different network flows.

Our testbed is in its infancy having been deployed for just over a month. While we are waiting to begin testbed wide deployments of ReSDN applications, such as TLSDeputy, we are beginning to look at the data we are collecting and considered what research questions could be answered. In

[1]The overview page is located at `https://resdn.cs.wpi.edu/ui/simpla`.

particular, we are interested in device identification using network activity. Figure 9.3 shows how we are considering using port activity as one means of identifying devices. Albeit a small subset of devices on the network, we can see that by grouping ports accessed into bins based on range clear patterns begin to emerge that distinguish different devices. Some devices are using a small subset of ports (likely IoT devices) and others use a wider variety of ports. In the future, ReSDN applications could use the information for automatically determining what applications should be in effect on a per device basis.

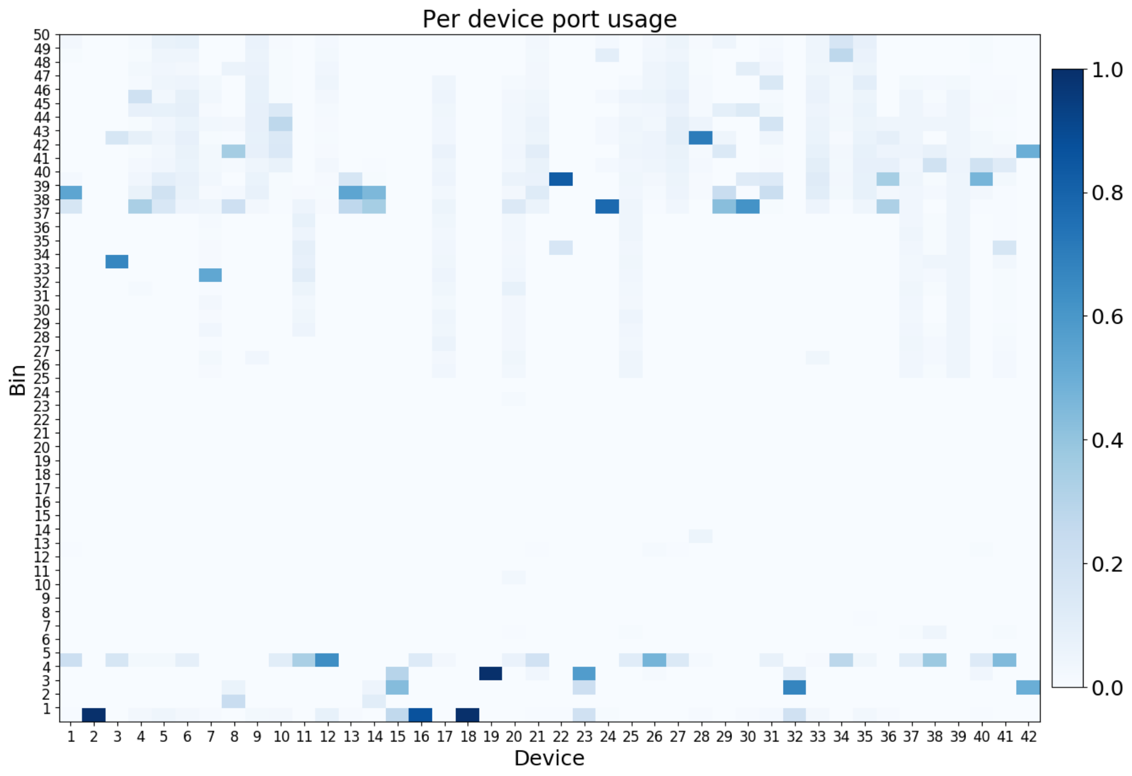


Figure 9.3: We group network port usage of outgoing connections into 50 bins that evenly divide all network ports for 42 devices on the ReSDN testbed. Devices that appear on multiple networks are shown multiple times.

9.5 Summary

In Chapter 5, we introduce the ReSDN infrastructure. Since its introduction, the limitations of a small deployment have become more apparent. This chapter has shown our efforts in moving towards a large scale ReSDN testbed. With this testbed, we are able to alleviate concerns of the homogeneity of a single home deployment. We have managed to successfully deploy ReSDN routers in three homes across two states. In deploying ReSDN routers, we have addressed several real-world considerations that are often overlooked in research. As part of our IRB protocol, we have provided solutions for both maintaining security of the router and enabling multiple privacy preserving techniques configurable by participants. The testbed is in its infancy, but we are already beginning to utilize the data collected and consider the a new set of research questions it will help to answer, such as device identification.

Chapter 10

Future Work

Our efforts in realizing host-based SDN in the enterprise and outsourcing residential network security and control to the cloud open the door for future research in these areas.

10.1 Enterprise Host-based SDN

While we believe there are serveral future research efforts in the enterprise, we believe that integrating forensics is a near term research effort for host-based SDN that is both beneficial and impactful.

10.1.1 Forensics

By fusing network, system, and GUI information together, we are able to understand a more complete story about user interactions and resulting network traffic. This information may be particularly useful in forensically reconstructing events that result in compromise. The goal of our GUI signature approach as discussed in Chapter 4 was to detect and block connections that do not follow legitimate paths through a GUI interface. Unfortunately, ambiguity in GUI applications may allow specially crafted malware to execute and create connections. Once detected, having information related to the events leading up to the compromise may be invaluable to determining how the compromise occurred (i.e., the attack vector), what steps to take to recover, and importantly, how to prevent the attack in the future. As an example, malicious emails are frequently received by organizations and may contain, for example, attachments contain malware [20]. Empowering a forensic analyst with a complete story resulting in the compromise may speed up the timeline of recovery and prevention.

10.1.2 User-in-the-Loop Security

Shirley *et al.* [158] suggested that understanding a user's intention could help in making decisions about whether or not file access should be granted to an application. Relating intent to network usage, an ideal scenario would allow a user to easily inform a network operator about the intent behind using an application. Such a scenario would allow both network operators to achieve their security goals while taking into account user goals and preferences. This approach is in contrast to today's network policy enforcement which frequently excludes end-users from the decision making process.

Future work would see us extend our host-based agent to include a feedback loop with the user to allow explicit network exceptions and tailor policies for each user. Because our approach gathers different sources of information (network, system, and GUI), we can better inform users of why particular connections were blocked and accordingly, allow them to make explicit exceptions. Network operators would also benefit in better understanding what requests are being made and quickly make decisions on whether or not to allow communication.

10.2 Residential SDN

Our ReSDN testbed deployment described in Chapter 9 has created a foundation for addressing security in the Internet of Things on residential networks in the future.

10.2.1 Advancing the IoT Security

Our residential SDN work has looked at deploying middleboxes network-wide to help protect network devices. While traditional network security solutions may be applied in a whole-home fashion, the Internet of Things (IoT) will require a more focused security effort. The consequences of poor IoT security have led to relatively small threats such as a network attack that causes printer ink to be exhausted [40] to reports of a large scale botnet with over 100,000 IoT devices including refrigerators, televisions, multimedia centers, and routers being used by attackers to send spam and phishing emails [39].

10.2.2 IoT Fingerprints and Profiles

Using our testbed, we envision being able to believe a database of IoT device fingerprints and profiles. Importantly, we seek to not require end-user maintenance or manufacturer support. These efforts will both help prevent and detect compromised devices.

Fingerprinting Devices

Device fingerprints will help us to precisely and automatically determine various devices connected to a home network with requiring input from the user. We plan to build fingerprints by closely monitoring network traffic behavior and characteristics.

Using our ReSDN router, we will be able to leverage information such as the media access control (MAC) address of devices to determine the device manufacturer. This information may then be combined with passive data collection from our ReSDN testbed such as port usage as presented in Figure 9.3, TTL fingerprinting [66] and Multicast DNS [73]. We may also leverage active scanning techniques provided by tools such as nmap [26], which attempts to profile devices (e.g, using TCP options or open ports). These techniques are expected to provide a strong foundation for fingerprints given that devices have been found to have relatively low message diversity [169].

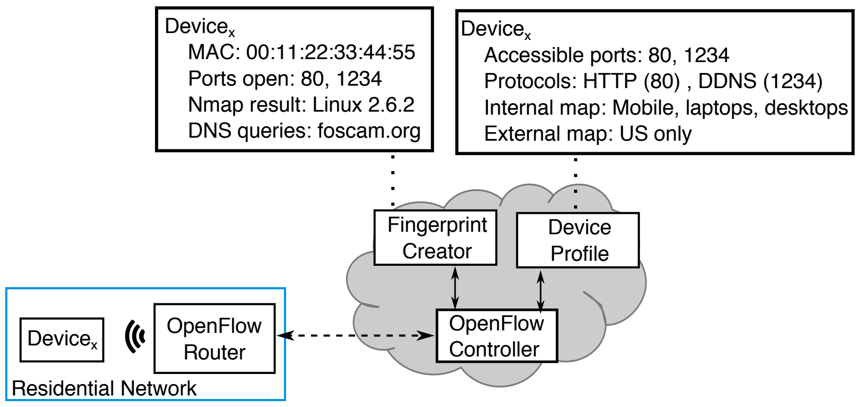


Figure 10.1: Scenario showing how a device within the home could be fingerprinted and subsequently have a profile built.

In Figure 10.1 we show an example fingerprint that could be built using a device's MAC address, ports open (e.g., discovered with nmap), the OS nmap believes to be running, and a list of domain names the device has queried. With automated analysis of the MAC address and some manual inspection of the data, we can conclude that Device$_x$ is a Foscam IP camera [13] operating wirelessly. This fingerprint could then be generally applied across all managed residential networks. A fingerprint can then be tied a specific profile.

Device Profiles

After building a device fingerprint, we then want to build a profile that enforces a security policy corresponding to the fingerprint. The profile will include knowledge of the device's capabilities and communication maps, in part, derived from the fingerprint. Continuing our example in Figure 10.1, we can build a profile that enforces which ports are accessible, what protocols run on those ports, and communication maps for internal and external access. The device may only be contacted on ports listed in the profile and may only use the protocols specified on each port. Although encryption presents some difficulty in monitoring application layer traffic for protocol or deep packet inspection analysis, techniques such as "peek and splice" [11] may be used to peek into encrypted traffic to check for security vulnerabilities.

The other component in our example profile is the internal and external communication maps. Devices that are compromised may attempt to "pivot" [30] inside the network to other devices by performing network scans. If the compromised devices attempts a large network scan, existing SDN anomaly detection techniques have been proven effective against scanning behavior [130]. However, an intelligent attacker could easily defeat such anomaly detection approaches. By applying profile communication maps internally and externally, we can help prevent pivoting by ensure certain devices inside the network are never able to communicate. One possibility is that a smart TV should never be able to communicate to an IP camera. The profile in Figure 10.1 ensures that only devices fingerprinted as mobile devices, laptops, or desktops are able to communicate to the IP camera. Furthermore, limiting the source of connections to US origins only, while simple,

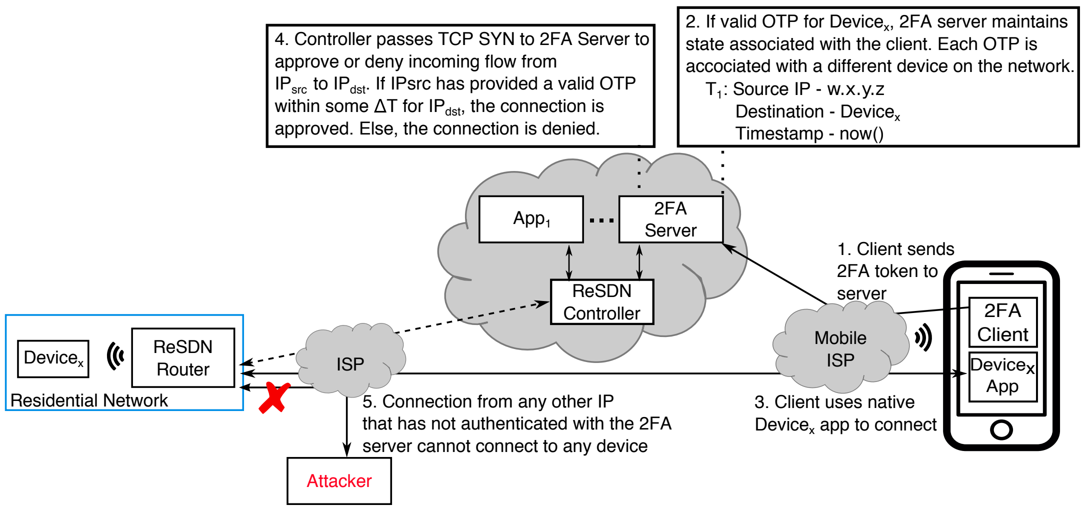

4. Controller passes TCP SYN to 2FA Server to approve or deny incoming flow from IPsrc to IPdst. If IPsrc has provided a valid OTP within some ΔT for IPdst, the connection is approved. Else, the connection is denied.
2. If valid OTP for Devicex, 2FA server maintains state associated with the client. Each OTP is accociated with a different device on the network.
T1: Source IP - w.x.y.z
Destination - Devicex
Timestamp - now()
App1
2FA Server
ReSDN Controller
1. Client sends 2FA token to server
2FA Client
Devicex App
Mobile ISP
Devicex
ReSDN Router
Residential Network
ISP
5. Connection from any other IP that has not authenticated with the 2FA server cannot connect to any device
3. Client uses native Devicex app to connect
Attacker

Figure 10.2: How 2FA can be generically applied to all IoT devices on a home network by centralizing the 2FA software and reversing the order of the first and second-factor authentication.

could actually reduce the volume of attack traffic received from the Internet by over 85% [5]. This isolation approach has similar goals to VLANs, but without the complexity or inflexibility associated with VLANs.

10.2.3 IoT Universal Two-factor Authentication

Another vision we have for advancing IoT security is universal two-factor authentication. Two-factor authentication (2FA) provides an added layer of security to authentication when logging in to accounts. The need for 2FA arose due to the fact that passwords may be used by anyone and may be easily guessed. 2FA requires an additional step to provide a one-time password (OTP) that is typically time sensitive (on the order of seconds).

Typically 2FA occurs after a providing a valid username and password to an authentication system. However, the system requiring the username and password may be a propietary IoT device on the network, and this system itself may be vulnerable, for example, to a buffer overflow attack [9]. We envision using 2FA such that the first factor is a one-time passcode and is given to a centralized authentication system before authenticating with a device. That is, ask for authentication of the one-time passcode before allowing a user to attempt a connection to supply a username and password to the IoT device. This process is in the reverse order of some popular 2FA mechanisms such as Google's. IoT devices currently cannot leverage the added security benefit of 2FA since the server (the IoT device) itself must support 2FA and be time synchronized with the connecting device. As such, our approach would first require the connecting device to submit the one-time passcode authentication using, for example, a mobile application running an authenticator application such as Google's Authenticator [18]. The client will use the mobile application to submit the passcode to an application communicating with the ReSDN controller. Once the application communicating with the controller receives the passcode, the user can launch the IoT device specific application, attempt a connection and supply a username and password. Without first using the passcode to the 2FA application, the ReSDN controller would drop the connection request and prevent any possible exploits to the IoT device. This process is shown in Figure 10.2.

Chapter 11

Summary and Concluding Remarks

11.1 Research Summary

In this dissertation we have both identified and addressed shortcomings in traditional enterprise and residential networks. In traditional enterprise networks, operators lack visibility into subnet traffic and have little information about network traffic that is seen. The research community introduced switch-based SDN to the enterprise with the hope of better addressing visibility. Unfortunately, scalability concerns hinder potential users of switch-based enterprise SDN. Our host-based SDN technique addresses both scalability concerns and then explores different methods of gathering information from an end-host to make more informed security decisions. We show how system-level information can be integrated into a well-established SDN protocol called OpenFlow. In addition to system information, we demonstrate how system context can be extended into the graphical user interface in order to understand how user interactions can serve as an endorsement for generated network traffic.

In residential networks where enterprise solutions cannot be directly applied, we present a residential cloud-based SDN solution that allows a remote third party to control a home network. With control outsourced, we present novel protections for residential users including a whole home proxy and a TLS verification and revocation solution. We find that our residential efforts are broadly applicable to users in the United States. Leveraging a geographically diverse participant pool, we show our ReSDN infrastructure is applicable to upwards of 90% of residential network users in the US. With this knowledge, we have begun an IRB-approved deployment of our ReSDN infrastructure to create a research testbed.

11.2 Concluding Remarks

Our work both in the enterprise and residential network environments accommodates practical deployment considerations while making novel contributions along the way. In part, our vision has been achievable by embracing the power software-defined networking. The SDN paradigm helps address network complexities by giving operators a centralized and programmatic view of a network. This allows us to think of networking problems as more of a traditional programming problem that is able to handle dynamic conditions rather than statically configured networks.

While completing this research, we have demonstrated various network attacks on enterprise and residential networks that, without our contributions, are difficult or impossible to detect with traditional approaches to security. Furthermore, our ReSDN infrastructure takes the burden off of residential network users and in turn allows third-party experts to better protect and secure the network. In completing this work, we hope that our discussion and evaluation has laid a solid foundation for future researchers to continue revolutionizing networks and network security with software-defined networking.

Bibliography

[1] 2016 state of the cloud survey. http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey.

[2] 2017 global adblock report. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf.

[3] About NOX. http://www.noxrepo.org/nox/about-nox/.

[4] Adblock pro. https://chrome.google.com/webstore/detail/adblock-pro/ocifcklkibdehekfnmflempfgjhbedch.

[5] Akamai's [state of the internet] / security q2 2015 report. https://www.stateoftheinternet.com/downloads/pdfs/2015-cloud-security-report-q2.pdf.

[6] Alexa top 1 million domains. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[7] Amazon mechanical turk. https://www.mturk.com/mturk/welcome.

[8] Barracuda web application firewall - client certificate validation using OCSP and CRLs. https://techlib.barracuda.com/waf/clientcertvalidation.

[9] Ca-2001-34 buffer overflow in system v derived login. https://www.cert.org/historical/advisories/CA-2001-34.cfm.

[10] CRLSets - the chromium projects. https://dev.chromium.org/Home/chromium-security/crlsets.

[11] Features: Sslpeekandsplice - squid web proxy wiki. http://wiki.squid-cache.org/Features/SslPeekAndSplice.

[12] Floodlight OpenFlow controller. http://www.projectfloodlight.org/floodlight/.

[13] Foscam - wireless IP cameras. http://foscam.us/.

[14] Geolite2 free downloadable databases - maxmind developer site. http://dev.maxmind.com/geoip/geoip2/geolite2/.

[15] Ghostery tracker browser extension — ghostery. https://www.ghostery.com/our-solutions/ghostery-browser-extension/.

[16] Google Chrome extension to measure page load time and display it in the toolbar. https://github.com/alex-vv/chrome-load-timer.

[17] The ICSI certificate notary. https://notary.icsi.berkeley.edu/#how-it-works.

[18] Install Google authenticator. https://support.google.com/accounts/answer/1066447.

[19] Internet connected devices surpass half a billion in U.S. homes, according to the NPD Group. https://www.npd.com/wps/portal/npd/us/news/press-releases/internet-connected-devices-surpass-half-a-billion-in-u-s-homes-according-to-the-npd-group/.

[20] Internet Security Threat Report VOLUME 21, APRIL 2016. https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf.

[21] Lenovo networking operating system. https://www.lenovo.com/images/products/system-x/pdfs/datasheets/lenovo_networking_operating_systems_ds.pdf.

[22] LibreSSL. http://www.libressl.org/.

[23] MITRE CVE - search results. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=foscam.

[24] Most of the US has no broadband competition at 25mbps, FCC chair says. http://arstechnica.com/business/2014/09/most-of-the-us-has-no-broadband-competition-at-25mbps-fcc-chair-says/.

[25] Network functions virtualisation an introduction, benefits, enablers, challenges & call for action. https://portal.etsi.org/nfv/nfv_white_paper.pdf.

[26] Nmap: the network mapper. https://nmap.org/.

[27] Notepad++ DLL hijack. https://wikileaks.org/ciav7p1/cms/page_26968090.html.

[28] Opendaylight platform. https://www.opendaylight.org.

[29] Operating system market share. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0.

[30] Pivoting - metasploit unleashed. https://www.offensive-security.com/metasploit-unleashed/pivoting/.

[31] POX. http://www.noxrepo.org/pox/about-pox.

[32] Ransomware: Information and prevention - Sophos community. https://community.sophos.com/kb/en-us/120797.

[33] Residential cord. https://wiki.opencord.org/display/CORD/Residential+CORD.

[34] Revoking intermediate certificates: Introducing OneCRL. https://blog.mozilla.org/security/2015/03/03/revoking-intermediate-certificates-introducing-onecrl/.

[35] Skype at 10: How an Estonian startup transformed itself (and the world). https://www.microsoft.com/en-us/stories/skype/skype-chapter-4-are-you-smoking.aspx.

[36] Skype audience stats - Microsoft advertising. https://advertising.microsoft.com/en-us/WWDocs/User/display/cl/brand_subproperty/1589/global/Skype-Audience-Stats.pdf.

[37] Skype ditched peer-to-peer supernodes for scalability, not surveillance. http://www.zdnet.com/article/skype-ditched-peer-to-peer-supernodes-for-scalability-not-surveillance/.

[38] Skype resolver. http://mostwantedhf.info/.

[39] Spam in the fridge: When the Internet of things misbehaves. http://www.economist.com/news/science-and-technology/21594955-when-internet-things-misbehaves-spam-fridge.

[40] Symantec - malware is causing network printers to print random ascii characters. https://support.symantec.com/en_US/article.tech190982.html.

[41] Top sites in United States. http://www.alexa.com/topsites/countries/US.

[42] Vulnerability note VU#656302 - Belkin Wemo home automation devices contain multiple vulnerabilities. https://www.kb.cert.org/vuls/id/656302.

[43] 2013 microsoft vulnerabilities study: Mitigating risk by removing user privileges. In *Avetco whitepaper*. 2013.

[44] OpenFlow specification version 1.3.0, June 2014.

[45] OpenWrt wireless freedom. https://openwrt.org, 2014.

[46] Proofpoint uncovers Internet of Things (IoT) cyberattack. http://investors.proofpoint.com/releasedetail.cfm?releaseid=819799, 2014.

[47] American time use survey - 2014 results. http://www.bls.gov/news.release/pdf/atus.pdf, June 2015.

[48] Electronic frontier foundation. https://www.eff.org/https-everywhere, 2015.

[49] Google safe browsing. https://www.google.com/transparencyreport/safebrowsing/, 2015.

[50] Internet traffic encryption. https://www.sandvine.com/trends/encryption.html, 2015.

[51] Let's encrypt. https://letsencrypt.org/, 2015.

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

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

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

[55] URLblacklist.com. http://urlblacklist.com/, 2015.

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

[57] Bernhard Amann, Matthias Vallentin, Seth Hall, and Robin Sommer. Revisiting SSL: A large-scale study of the Internet's most trusted protocol. Technical report, Citeseer, 2012.

[58] Jari Arkko and Pekka Nikander. Weak authentication: How to authenticate unknown principals without trusted parties. In *Security Protocols*, pages 5–19. Springer, 2002.

[59] S. Avallone, S. Guadagno, D. Emma, A. Pescape, and G. Ventre. D-ITG distributed Internet traffic generator. In *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pages 316–317, Sept 2004.

[60] Salman A Baset and Henning Schulzrinne. An analysis of the Skype peer-to-peer Internet telephony protocol. *arXiv preprint cs/0412017*, 2004.

[61] Alessandro Bassi, Martin Bauer, Martin Fiedler, Thorsten Kramp, Rob van Kranenburg, Sebastian Lange, and Stefan Meissner. Enabling things to talk, 2013.

[62] Arsany Basta, Wolfgang Kellerer, Marco Hoffmann, Hans Jochen Morper, and Klaus Hoffmann. Applying NFV and SDN to LTE mobile core gateways, the functions placement problem. In *Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications, and Challenges*, AllThingsCellular, New York, NY, USA, 2014. ACM.

[63] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behaviors. In *LEET*, 2009.

[64] Owen Bell, Mark Allman, and Benjamin Kuperman. On browser-level event logging. Technical report, 2012.

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

[66] Robert Beverly. A Robust Classifier for Passive TCP/IP Fingerprinting. In *Proceedings of the 5th Passive and Active Measurement (PAM) Workshop*, pages 158–167, April 2004.

[67] Wilson Naik Bhukya, Suneel Kumar Kommuru, and Atul Negi. Masquerade detection based upon GUI user profiling in linux systems. In *Annual Asian Computing Science Conference*, 2007.

[68] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, and Lingyu Wang. On the analysis of the Zeus botnet crimeware toolkit. In *Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on*, pages 31–38. IEEE, 2010.

[69] Dario Bonfiglio, Marco Mellia, Michela Meo, and Dario Rossi. Detailed analysis of Skype traffic. *Multimedia, IEEE Transactions on*, 11(1):117–127, 2009.

[70] Philip A Branch, Amiel Heyde, and Grenville J Armitage. Rapid identification of Skype traffic flows. In *Proceedings of the 18th international workshop on Network and operating systems support for digital audio and video*, pages 91–96. ACM, 2009.

[71] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy*, SP '14, pages 114–129, Washington, DC, USA, 2014. IEEE Computer Society.

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

[73] S. Cheshire and M. Krochmal. Multicast DNS. RFC 6762 (Proposed Standard), February 2013.

[74] S. Cheshire and M. Krochmal. NAT Port Mapping Protocol (NAT-PMP). RFC 6886 (Informational), April 2013.

[75] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, 2006.

[76] Aldo Cortesi. mitmproxy. `https://mitmproxy.org/`.

[77] Weidong Cui, Randy H Katz, and Wai-tian Tan. BINDER: An extrusion-based break-in detector for personal computers. In *USENIX Annual Technical Conference, General Track*, 2005.

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

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

[80] Italo Dacosta, Mustaque Ahamad, and Patrick Traynor. Trust no one else: Detecting MITM attacks against SSL/TLS without third-parties. In *Computer Security–ESORICS 2012*, pages 199–216. Springer, 2012.

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

[82] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176, 7465.

[83] John Dix. Clarifying the role of software-defined networking northbound APIs — network world. `http://www.networkworld.com/article/2165901/lan-wan/clarifying-the-role-of-software-defined-networking-northbound-apis.html`, 2013.

[84] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. Towards an elastic distributed SDN controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 7–12, New York, NY, USA, 2013. ACM.

[85] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. ETTM: A scalable fault tolerant network manager. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 85–98, Berkeley, CA, USA, 2011. USENIX Association.

[86] A. Doria, J. Hadi Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern. Forwarding and Control Element Separation (ForCES) Protocol Specification. RFC 5810 (Proposed Standard), March 2010. Updated by RFCs 7121, 7391.

[87] Will Dormann. Vulnerability note VU#720951 - OpenSSL TLS heartbeat extension read overflow discloses sensitive information. `https://www.kb.cert.org/vuls/id/720951`.

[88] Sven Ehlert, Sandrine Petgang, Thomas Magedanz, and Dorgham Sisalem. Analysis and signature of Skype VoIP session traffic. *4th IASTED International*, 2006.

[89] David Erickson. The Beacon OpenFlow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2013.

[90] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 533–546, Berkeley, CA, USA, 2014. USENIX Association.

[91] Nick Feamster. Outsourcing home network security. In *Proceedings of the 2010 ACM SIGCOMM Workshop on Home Networks*, HomeNets '10, pages 37–42, New York, NY, USA, 2010. ACM.

[92] Andrew D Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 327–338. ACM, 2013.

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

[94] Open Networking Foundation. OpenFlow 1.0 switch specification. `archive.openflow.org/documents/openflow-spec-v1.0.0.pdf`, 2014.

[95] T. Freeman, R. Housley, A. Malpani, D. Cooper, and W. Polk. Server-Based Certificate Validation Protocol (SCVP). RFC 5055 (Proposed Standard), December 2007.

[96] Carrie Gates, Michael Collins, Michael Duggan, Andrew Kompanek, and Mark Thomas. More netflow tools for performance and security. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04.

[97] Carrie Gates, Michael Collins, Michael Duggan, Andrew Kompanek, and Mark Thomas. More netflow tools for performance and security. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, pages 121–132, Berkeley, CA, USA, 2004. USENIX Association.

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

[99] Aaron Gember, Robert Grandl, Junaid Khalid, and Aditya Akella. Design and implementation of a framework for software-defined middlebox networking. *SIGCOMM Comput. Commun. Rev.*, 43(4):467–468, August 2013.

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

[101] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[102] H.H. Gharakheili, J. Bass, L. Exton, and V. Sivaraman. Personalizing the home network experience using cloud-based SDN. In *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*, pages 1–6, June 2014.

[103] Glen Gibb, Hongyi Zeng, and Nick McKeown. Outsourcing network functionality. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, pages 73–78, 2012.

[104] Dan Goodin. Man-in-the-middle attack on Vizio TVs coughs up owners' viewing habits. `http://arstechnica.com/security/2015/11/man-in-the-middle-attack-on-vizio-tvs-coughs-up-owners-viewing-habits/`. (Accessed 03-05-2016).

[105] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting SSL usage in applications with SSLint. In *Security and Privacy (SP)*, pages 519–534. IEEE, 2015.

[106] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 7–12. ACM, 2012.

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

[108] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, New York, NY, USA, 2009. ACM.

[109] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kaafar. TLS in the wild: an Internet-wide analysis of TLS-based protocols for electronic communication. *arXiv preprint arXiv:1511.00341*, 2015.

[110] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. The SSL landscape: A thorough analysis of the x.509 PKI using active and passive measurements. In *Internet Measurement Conference*, pages 427–444, New York, NY, USA, 2011. ACM.

[111] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *Proceedings of 2015 Annual Network and Distributed System Security Symposium (NDSS'15)*, February 2015.

[112] Michael Horowitz. A router firmware update goes bad. `http://www.computerworld.com/article/2692514/a-router-firmware-update-goes-bad.html`, October 2014.

[113] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged SSL certificates in the wild. In *Security and Privacy (SP)*, pages 83–97. IEEE, 2014.

[114] Yeongjin Jang, Simon P Chung, Bryan D Payne, and Wenke Lee. Gyrus: A framework for user-intent monitoring of text-based networked applications. In *NDSS*, 2014.

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

[116] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *Communications Magazine, IEEE*, 51(2):114–119, 2013.

[117] Keith Kirkpatrick. Software-defined networking. *Communications of the ACM*, 56(9):16–19, 2013.

[118] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[119] Maciej Kuzniar, Peter Peresini, and Dejan Kostic. What you need to know about sdn flow tables. In *Lecture Notes in Computer Science (LNCS)*, number EPFL-CONF-204742, 2015.

[120] Jonghoon Kwon, Jehyun Lee, and Heejo Lee. Hidden bot detection by tracing non-human generated traffic at the zombie host. In *International Conference on Information Security Practice and Experience*, 2011.

[121] Kiran Lakkaraju, William Yurcik, and Adam J. Lee. NVisionIP: Netflow visualizations of system state for security situational awareness. In *ACM Workshop on Visualization and Data Mining for Computer Security*, October 2004.

[122] Stevens Le Blond, Chao Zhang, Arnaud Legout, Keith Ross, and Walid Dabbous. I know where you are and what you are sharing: exploiting P2P communications to invade users' privacy. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 45–60. ACM, 2011.

[123] Minseok Lee, Younggi Kim, and Younghee Lee. A home cloud-based home network auto-configuration using SDN. In *IEEE International Conference onNetworking, Sensing and Control (ICNSC)*, 2015.

[124] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An end-to-end measurement of certificate revocation in the webs pki. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, pages 183–196. ACM, 2015.

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

[126] Long Lu, Vinod Yegneswaran, Phillip Porras, and Wenke Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 440–450. ACM, 2010.

[127] Gregor Maier, Anja Feldmann, Vern Paxson, and Mark Allman. On dominant characteristics of residential broadband Internet traffic. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, 2009.

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

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

[130] SyedAkbar Mehdi, Junaid Khalid, and SyedAli Khayam. Revisiting traffic anomaly detection using software defined networking. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, volume 6961 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011.

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

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

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

[134] R. Mortier, T. Rodden, T. Lodge, D. McAuley, C. Rotsos, A.W. Moore, A. Koliousis, and J. Sventek. Control and understanding: Owning your home network. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pages 1–10, Jan 2012.

[135] Mohamed E. Najd and Craig A. Shue. DeepContext: An OpenFlow-compatible, host-based SDN for enterprise networks. Pending.

[136] Jad Naous, Ryan Stutsman, David Mazieres, Nick McKeown, and Nickolai Zeldovich. Delegating network security with more information. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 19–26. ACM, 2009.

[137] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The cost of the S in HTTPS. In *ACM Conference on Emerging Networking Experiments and Technologies*, pages 133–140. ACM, 2014.

[138] Colin Neagle. Smart refrigerator hack exposes Gmail account credentials. `http://www.networkworld.com/article/2976270/internet-of-things/smart-refrigerator-hack-exposes-gmail-login-credentials.html`.

[139] Bao N Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 2014.

[140] Gabriele Paolacci, Jesse Chandler, and Panagiotis G Ipeirotis. Running experiments on amazon mechanical turk. *Judgment and Decision making*, 2010.

[141] Bryan Parno, Zongwei Zhou, and Adrian Perrig. Using trustworthy host-based information in the network. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12, pages 33–44, New York, NY, USA, 2012. ACM.

[142] Marcell Perényi, András Gefferth, Trang Dinh Dang, and Sándor Molnár. Skype traffic identification. In *Global Telecommunications Conference, 2007. GLOBECOM'07. IEEE*, pages 399–404. IEEE, 2007.

[143] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[144] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. Extending networking into the virtualization layer. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2009.

[145] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of Open vSwitch. In *USENIX symposium on networked systems design and implementation*, 2015.

[146] Po Qi, Cuilan Du, Yan Ren, and Yibo Xue. The secrets of Skype login. 2013.

[147] Jennifer Rexford and Constantine Dovrolis. Future Internet architecture: clean-slate versus evolutionary research. *Communications of the ACM*, 2010.

[148] M. Richardson and D. H. Redelmeier. Opportunistic encryption using the Internet key exchange (IKE). IETF RFC 4322, December 2005.

[149] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Security and privacy (SP), 2012 IEEE Symposium on*, 2012.

[150] Daniel Roethlisberger. SSLsplit - transparent SSL/TLS interception). `https://www.roe.ch/SSLsplit`.

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

[152] J. Schlyter and W. Griffin. Using DNS to securely publish secure shell (SSH) key fingerprints. IETF RFC 4255, January 2006.

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

[154] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM.

[155] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. 2009.

[156] Seungwon Shin, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, Guofei Gu, and Mabry Tyson. Fresco: Modular composable security services for software-defined networks. In *NDSS*, 2013.

[157] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 78–89, New York, NY, USA, 2014. ACM.

[158] Jeffrey Shirley and David Evans. The user is not the enemy: Fighting malware by tracking user intentions. In *Proceedings of the 2008 New Security Paradigms Workshop*, NSPW '08.

[159] C.A. Shue, A.J. Kalafut, and M. Gupta. Abnormally malicious autonomous systems and their Internet connectivity. *Networking, IEEE/ACM Transactions on*, 20(1):220–230, Feb 2012.

[160] Eric Smith. Global broadband and WLAN (Wi-Fi) networked households forecast 2009-2018. `https://www.strategyanalytics.com/access-services/devices/connected-home/consumer-electronics/reports/report-detail/global-broadband-and-wlan-(wi-fi)-networked-households-forecast-2009-2018#.VlzX6d-rSV4`.

[161] Murugiah Souppaya and Karen Scarfone. *Guide to Malware Incident Prevention and Handling for Desktops and Laptops.* U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology Gaithersburg, MD, 2013.

[162] Marc Stiegler, Alan H Karp, Ka-Ping Yee, Tyler Close, and Mark S Miller. Polaris: virus-safe computing for windows xp. *Communications of the ACM*, 2006.

[163] Srikanth Sundaresan, Sam Burnett, Nick Feamster, and Walter De Donato. BISmark: A testbed for deploying measurements and applications in broadband access networks. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 383–394, Berkeley, CA, USA, 2014. USENIX Association.

[164] Srikanth Sundaresan, Walter De Donato, Nick Feamster, Renata Teixeira, Sam Crawford, and Antonio Pescapè. Broadband Internet performance: a view from the gateway. In *ACM SIGCOMM computer communication review*. ACM, 2011.

[165] Srikanth Sundaresan, Nick Feamster, and Renata Teixeira. Measuring the performance of user traffic in home wireless networks. In *International Conference on Passive and Active Network Measurement*, 2015.

[166] Curtis Taylor, Douglas MacFarland, Doran Smestad, and Craig Shue. Contextual, flow-based access control with scalable host-based SDN techniques. In *Computer Communications (INFOCOM), 2016 IEEE Conference on*. IEEE, 2016.

[167] Curtis R. Taylor and Craig A. Shue. Validating security protocols with cloud-based middleboxes. In *IEEE Conference on Communications and Network Security (CNS)*, 2016.

[168] Curtis R. Taylor, Craig A. Shue, and Mohamed E. Najd. Whole home proxies: Bringing enterprise-grade security to residential networks. In *IEEE International Conference on Communications (ICC)*, May 2016.

[169] Spyridon Tompros. nternet-of-things architecture (IOT-A) project deliverable d3.1 - initial M2M API analysis. `http://www.iot-a.eu/public/public-documents/d3.1`.

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

[171] Miles Tracy, Wayne Jansen, Karen Scarfone, and Theodore Winogard. Guidelines on securing public web servers. *NIST Special Publication 800-44, Version 2*, page 142, September 2007.

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

[173] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM, 2002.

[174] Robert J. Walls, Eric D. Kilmer, Nathaniel Lageman, and Patrick D. McDaniel. Measuring the impact and perception of acceptable advertisements. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, IMC '15, pages 107–120, New York, NY, USA, 2015. ACM.

[175] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. Speeding up web page loads with shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation*, 2016.

[176] Huijun Xiong, Danfeng Yao, and Zhibin Zhang. Storytelling security: User-intention based traffic sanitization. 2010.

[177] Yiannis Yiakoumis, Sachin Katti, Te-Yuan Huang, Nick McKeown, Kok-Kiong Yap, and Ramesh Johari. Putting home users in charge of their network. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 1114–1119, New York, NY, USA, 2012. ACM.

[178] Yiannis Yiakoumis, Kok-Kiong Yap, Sachin Katti, Guru Parulkar, and Nick McKeown. Slicing home networks. In *Proceedings of the 2nd ACM SIGCOMM workshop on Home networks*, pages 1–6. ACM, 2011.

[179] Yiannis Yiakoumis, Kok-Kiong Yap, Sachin Katti, Guru Parulkar, and Nick McKeown. Slicing home networks. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Home Networks*, HomeNets '11, pages 1–6, New York, NY, USA, 2011. ACM.

[180] Zhenlong Yuan, Cuilan Du, Xiaoxian Chen, Dawei Wang, and Yibo Xue. Skytracer: Towards fine-grained identification for Skype traffic via sequence signatures. In *Computing, Networking and Communications (ICNC), 2014 International Conference on*, pages 1–5. IEEE, 2014.

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

[182] Hao Zhang, Danfeng Daphne Yao, and Naren Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014.

[183] Liang Zhu, Johanna Amann, and John Heidemann. Measuring the latency and pervasiveness of tls certificate revocation. In *Passive and Active Measurement*, pages 16–29. Springer, 2016.