ABSTRACT

GOUTAM, SANKET. Hestia: Simple Least Privilege Network Policies for Smart Homes. (Under the direction of William Enck and Bradley Reaves).

The long-awaited smart home revolution has arrived, and with it comes the challenge of managing dozens of potentially vulnerable network devices by average users. While research has developed techniques to fingerprint these devices, and even provide for sophisticated network access control models, such techniques are too complex for end users to manage, require sophisticated systems or unavailable public device descriptions, and proposed network policies have not been tested against real device behaviors. As a result, none of these solutions are available to users today.

To address these shortcomings, we present Hestia, a mechanism to enforce simple-but-effective network isolation policies. Hestia segments the network into just two device categories: controllers (e.g., Smart Hubs) and non-controllers (e.g., motion sensors and smart lightbulbs). The key insight (validated with a large IoT dataset) is that non-controllers only connect to cloud endpoints and controller devices, and practically never to each other over IP networks. This means that non-controllers can be isolated from each other without preventing functionality. Perhaps more importantly, smart home owners need only specify which devices are controllers. We develop a prototype and show negligible performance overhead resulting from the increased isolation. Hestia drastically improves smart home security without complex, unwieldy policies or lengthy learning of device behaviors.

Hestia: Simple Least Privilege Network Policies for Smart Homes

by
Sanket Goutam

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

William Enck Co-chair of Advisory Committee	Bradley Reaves Co-chair of Advisory Committee

Anupam Das

DEDICATION

To mom, dad, and my grandparents for supporting the whims of a curious mind.

BIOGRAPHY

The author hails from a small industrial town in Odisha, India. Prior to joining the graduate program at North Carolina State University, the author earned his Bachelor of Technology (B.Tech) in Computer Science & Engineering from VIT University in 2017.

ACKNOWLEDGEMENTS

This dissertation is an outcome of not just my academic undertakings, but also a series of compromises, endurance, and constant support from my friends and family over the course of two years. At the culmination of this journey I find myself guilty of not repeatedly expressing my gratitude to all those who have played a huge role in this endeavour.

First and foremost, I express my deepest gratitude to my advisors Dr. William Enck and Dr. Bradley Reaves for their instrumental help in guiding my research. I find myself incredibly fortunate to have the opportunity to work with them both, and in process acquire invaluable lessons on being an effective academic. I cherish this experience greatly, and find myself at an advantage towards my eventual goal of pursuing a doctoral degree. I would also like to thank Dr. Anupam Das for taking the time to be on my committee, and for being supportive of my research. I also extend my thanks to several members of the WSPR lab, especially Terrence O' Connor, Benjamin Andow, and Issac Polinsky, for helping me formulate my research ideas and providing their valuable insights to effectively direct my research.

Finally, I thank my family and my friends for always being supportive of my decisions and providing the encouragement at all times. I would like to specifically convey my deepest gratitude to some of my oldest friends, Sumeet, Mucron, Nilav, Prachi, Fouzia, for being the anchor throughout all chapters of my life. I thank Abhayjeet and Shivam for their numerous pep talks and I also thank Athishay, my roommate for two years and friend for longer, for putting up with me at all times. It would not have been possible without the support from all these people.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Internet connected home devices, once a niche product category, are now a normal part of consumers' lives. In the United States alone, 33.2% of homes have at least one smart device, and this number is expected to grow to 53.9% in the next four years [Sta18]. The value of these devices comes in part from their ability to be automated and controlled in tandem from single interfaces like smart hubs. Not only do these devices provide the convenience of remotely monitoring temperature and carbon monoxide sensors, accessing video surveillance, tracking pets' movements, and remotely locking doors, they permit recipes like scheduling lights, music, and coffee for breakfast time. As a result, devices of varying computational ability manufactured by different vendors are filling homes, in some

cases tripling the number of Internet-connected devices in the home.

These devices, individually and in toto, create significant security concerns. Not only are they deployed to perform sensitive tasks (e.g., cameras monitoring children) but also play important safety roles (e.g., door locks, smoke detection). Moreover, repeated security flaws leading to severe operational failure [Cim18], mass exploitation [Ant17], and use of devices as a pivot to attack other networked devices [Wei18] mean trust in the average device is largely misplaced. Popular devices have been found to be exploitable from the local network [Smi14; SH13], creating the very real threat that a single compromised device may then attack other local devices. Prior incidents [Wei18; Waq17] where IoT devices have been exploited to compromise other systems on the same network make this threat very clear. The fact that these devices are collections of diverse hardware and software complicates any and every solution. The sheer number of devices alone currently or soon-to-be deployed means that users would be overwhelmed with managing any host-based security solution.

Previous attempts addressing the security of IoT systems have focused on traditional mechanisms like providing intrusion detection systems [Raz13], lightweight authentication schemes [HR15], behavioral fingerprinting of IoT [Bez18; FS18a; Haf17], whitelisting connections at the gateway router [Bar18], or providing new security abstractions by authenticating application data flows on devices [Yu15; Dem17; Fer16a; Tia17; Jia17]. Unfortunately these mechanisms either require a redesign of legacy devices, are not scalable for the constantly evolving smart home ecosystem, or focus solely on selected platforms. Recently, Schuster et al. [Sch18] focused on providing access control to devices based on the context of their environmental situation, however such complex systems require implementation in the entire IoT stack of different frameworks by all manufacturers, thus making it not readily deployable in smart homes today.

1.1 Thesis statement

We hypothesized that a simple solution can be provided by characterizing the functionality that each device exhibits in a users' smart home. Smart home devices, in general, are designed for very specific purposes implying that under ideal conditions their usage in a smart home does not deviate. We also identified that these devices either are trivial IoT sensors and actuators which need control commands to function, or are automation devices (like smart hubs) that send these control commands over the network. Putting it in simpler terms, we could say that all smart home devices either have the functionality to control others or do not. So by extending this analogy we can infer that, by design, trivial IoT devices, like smart light bulbs, can not receive control commands from each other. The functionality of sending commands to a light bulb is attributed to only a specific smart phone or a lighting automation hub, and thus only these devices should be able to communicate with the light bulb. This understanding of device interactions in a smart home is quite interesting because it implies that network access policies could be created leveraging the functionality attributed to each device.

We define our Thesis statement as follows: "Smart home devices mostly exhibit limited and predictable communication patterns on a local network, providing an opportunity to define simple and practical least privilege policy for network access."

This thesis proposes a new system, Hestia, whose goal is to reduce risk of compromise in smart homes by implementing a least-privilege network policy. We first note that smart home networks consist of a few controller devices, like automation hubs and personal assistants that provide user interfaces, and the remaining non-controller devices that perform sensing, monitoring, or actuating duties. We find that almost without exception, non-controller devices only communicate with controllers and the cloud, and never directly

from non-controller device-to-device. The key insight behind Hestia is that we can isolate individual smart home non-controller devices to only communicate with controllers and the cloud and no other devices. This provides a drastic reduction in possible communication paths — approximating least-privilege access to the network for non-controller devices. Configuration is then incredibly straightforward: users need only specify which devices are controllers.

1.2 Contributions

Our work makes the following contributions:

- We demonstrate that our controller–non-controller dichotomy is robust using a large public dataset of over 40 smart home devices [Alr19].
- We then construct a prototype of Hestia as a SDN application for Open vSwitch; in so doing, we address policy management, enforcement, and device discovery.
- We demonstrate that Hestia provides significantly enhanced isolation with minimal performance impacts compared to a stock wireless access point.

We are not the first to explore network policies to constrain IoT device behavior. However, prior work either proposes enforcement mechanisms without testing or validating policies [Mie17], or requires fine-grained policies [Bar18] that require significant investment to specify or discover using (fallible) learning methods. This work provides a validated policy that can be deployed *today* with minimal effort.

1.3 Organization

The remainder of this thesis is structured as follows.

Chater 2 provides a background on the technologies used in this work, and also identifies the key aspects contributing to the diversity in a smart home that makes this work challenging. Chapter 3 provides the related work in the domain, identifying their contributions and shortcomings and how our work differentiates from them. In Chapter 4, we provide our hypothesis on the smart home ecosystem and demonstrate its validation against a large IoT data set. We explain our hypothesis by providing a use case for Hestia and define our threat model in respect to the standard smart home configuration. Chapter 5 provides an overview of our system, and describes the design of Hestia. In Chapter 6 we summarize the performance evaluation of Hestia through experiments to demonstrate its readiness for deployment in a smart home.

CHAPTER

2

BACKGROUND

The first section of this chapter introduces the smart home ecosystem and provides an insight on the IoT design stack, followed by the various communication protocols used by smart home devices attributing to their huge diveristy, and then identifies the key aspects of a connected home which we have targetted in this work. The second section in this chapter introduces Software Defined Networking (SDN), which is a new paradigm in networking with enhanced features and capabilities that are not easily found in traditional networks. We leverage SDN primitives to design our solution for a smart home environment.

2.1 Smart Home Ecosystem

The term "Smart Home" refers to a household where home automation or domotics [Hil15a] is used to seamlessly network devices and appliances to provide users with a complete control over all aspects of their home. It is commonly used to define a residence comprising of appliances, lighting, heating, air conditioning, TVs, computers, entertainment audio & video systems, security, and camera systems that are capable of communicating with one another and can be controlled remotely by a time schedule. Home automation has been around for many decades in the form of lighting and simple appliance control. But the advancements in wireless IoT network protocols have caught up with the idea of the interconnected world, providing automation at our fingertips on smartphones, and through voice commands to virtual assistants. Interfacing of IoT devices in the household has offered us the convenience of using automation recipes to remotely trigger lights, open window blinds, and play music all simultaneously with the morning alarm.

2.1.1 IoT stack

The majority of IoT devices - from smart plugs to thermostats, door sensors to security cameras - all developed by different vendors come with their own systems and connections. This is because of the diverse range of development frameworks that are supported by an IoT stack. A generic IoT stack includes three layers: application, transport, and sensing [Yaq17]. However, a more detailed IoT deployment is usually adopted which includes five layers [Kha12]:

- Perception layer: also known as 'Device layer', includes sensor devices and physical objects

- Network layer: also known as 'Transmission layer'. It is responsible for securely transferring data from sensing devices to the information processing system
- Middleware layer: This layer is responsible for service management and provision of interconnection to the system database. It processes information, performs ubiquitous computations, and makes automatic decisions based on the outputs
- Application layer: This layer provides global management of the provided applications considering the objects information which was processed from the middleware layer
- Business layer: This layer is responsible for the management of the whole IoT system, including services, and applications

Each of these layers can support a wide variety of protocols and designs, thus making the smart home ecosystem even more heterogeneous.

2.1.2 Communication Protocols

Smart home devices cover a huge variety of appliances and use cases that range from a single constrained device up to massive cross platform deployments. Numerous legacy and emerging communication protocols are used by vendors that allow devices and servers to talk to each other in new, more interconnected ways. We describe some of the more common protocols that are used by smart home devices to interact with each other.

1. **Infrared** - This method of communication is used by simple and reliable products, normally offering one-way communication. Mostly used in TV remote controls.
2. **Bluetooth** - Bluetooth is another low energy communication protocol that is widely used in smart home devices. Most inexpensive devices like smart light bulbs, use bluetooth to communicate with the users' smart phone.

3. **Ethernet** - Some IoT devices can be connected to a home network via an ethernet interface. However these devices generally are Hubs, or workstations. Most commodity IoT hardwares do not provide an ethernet interface for connection.

4. **WiFi** - The most common form of wireless communication between devices in a smart home. Since all IoT devices require connection to their cloud endpoints for effective functionality, they need to be connected to the home wireless access point.

5. **Thread** - Thread is a wireless protocol developed by a group of companies including Nest, Samsung, Qualcomm, and OSRAM. It is designed to allow the devices in its protocol to communicate even when the WiFi network goes down.

6. **Zigbee** - Zigbee is a wireless protocol which operates in a mesh network. It relays a signal from one device to another, strengthening and expanding the network. It is mostly built in dimmers, door locks, thermostats, and other home appliances. WeMo and Phillips devices use Zigbee as the primary communication protocol.

7. **Z-Wave** - Similar to Zigbee, Z-Wave is also an open source mesh network protocol. The only major difference between the two is the data throughput, as Z-wave is roughly 6 times slower than Zigbee, however requires less energy than Zigbee to cover the same range. Samsung SmartThings and Lowes Iris use Z-Wave as their communication protocol.

8. **Insteon** - The Insteon protocol is a hybrid of wireless and wired technologies. It operates on a dual-mesh network that uses both wireless and hardwired communication to support communication between devices in a large mesh networks.

2.1.3 Home automation platforms

Each of these protocols mentioned above have their own appeal for a specific use case, leading manufacturers to produce a line of smart home products that use the same communication protocols. In general, these protocols are still not directly compatible with each other thereby limiting the inter-vendor compatibility of devices for end users. So, in order to foster a connected ecosystem of smart home devices and encourage community based software development practices, many smart home platforms have been developed. Some of the most popular smart home platforms include Samsung's SmartThings [Sam18], Apple's HomeKit [App18], Google's Brillo and Weave [CNE15], Vera Control's Vera [Ver18], and AllSeen Alliance's AllJoyn [Fou18]. These platforms act as one unifying box enabling users to pair the functionality of different devices from different vendors. These platforms, or as commonly known as automation hubs, support multiple communication protocols and connect all of users devices to a central hub, and then provide a single interface (in the form of a smart phone app) for the user to control all their devices.

2.1.4 Device discovery

Interfacing multiple devices with an automation hub, or even pairing the functionality of devices together directly requires service discovery. Service discovery is the distinguishing factor between an IoT network and a typical Internet network. These smart home devices provide their functionality by hosting a particular service or application and then advertising it through several different protocols. Any other device on the same network, wishing to avail these services has to listen for the service advertisement packets using the same protocol. Some of the most common discovery protocols used by smart home devices are:

- **mDNS** - or multicast Domain Name System. It resolves host names to IP addresses

within small networks that do not include a name server

- **Physical Web** - It enables users to see a list of URLs being broadcasted by objects in the environment with a Bluetooth Low Energy (BLE) beacon.

- **HyperCat** - This is an open, lightweight JSON-based hypermedia catalogue format for exposing collections of URIs.

- **UPnP** - or Universal Plug and Play. It is now managed by the Open Connectivity Foundation, and is a set of networking protocols that permit networked devices to seamlessly discover each other's presence on the network and establish functional network services for data sharing, communications, and entertainment.

Studying and understanding the various components of a smart home, from the design architectures of IoT devices to the communication protocols required for device interaction, has helped in identifying the design goals of Hestia. The goal of this thesis is to provide an effective network abstraction for smart homes that can be deployed readily by users. Thus it is imperative that Hestia supports the heterogeneity of device architectures, communication protocols, legacy systems, and achieves it with minimal overhead on the users.

2.2 Software-Defined Networking

According to the Open Networking Foundation, Software-Defined Networking (SDN) is the emerging network architecture that provides a dynamic, manageable, cost-effective, and adaptable way of dealing with the dynamic nature of today's applications. In simpler terms, SDN provides the capability of decoupling the network control and forwarding functions enabling the network control to become directly programmable and the underlying infras-

tructure to be abstracted for applications and network services. The distinction between control plane and data plane can be expalined as follows:

- The **control plane** is the administration layer of the network. It allows setting up of the packet processing rules, and then establishes the whole network switching policy.
- The **data plane** encompasses the application of those rules defined on control planes. This layer performs the actual packet processing. When some packets require some particular, more complex processing, they can be handled to the control plane, where the decision regarding this packet will be taken by the SDN applications.

2.2.1 SDN architecture

The SDN architecture, as shown in Figure 2.1, generally has three components or groups of functionality:

- **SDN Applications:** These are the programs that communicate behaviors and needed resources with the SDN controller via application programming interfaces (APIs). In addition, these applications also provide an abstracted view of the network by collecting information from the controller for decision-making purposes. Applications may include networking management, analytics, or business applications used to run large data centers.
- **SDN Controller:** The controller is a logical entity that receives instructions or requirements from the SDN application layer and relays them to the underlying networking components. The controller also extracts information about the network from the hardware devices and communicates back to the SDN applications with an abstract view of the network.

- **Networking devices:** These are the physical devices that control the forwarding and data processing capabilities for the network. This includes forwarding and processing of the data path.

The APIs shown in the Figure 2.1 are often referred to as northbound and southbound interfaces, defining the communication between the applications, controllers, and networking systems. A Northbound interface is the connection between a SDN controller and its supporting applications, whereas the Southbound interface denotes the conntection between a controller and the physical networking hardware. Since SDN is a virtualized architecture, these elements do not have to be physically located in the same place.

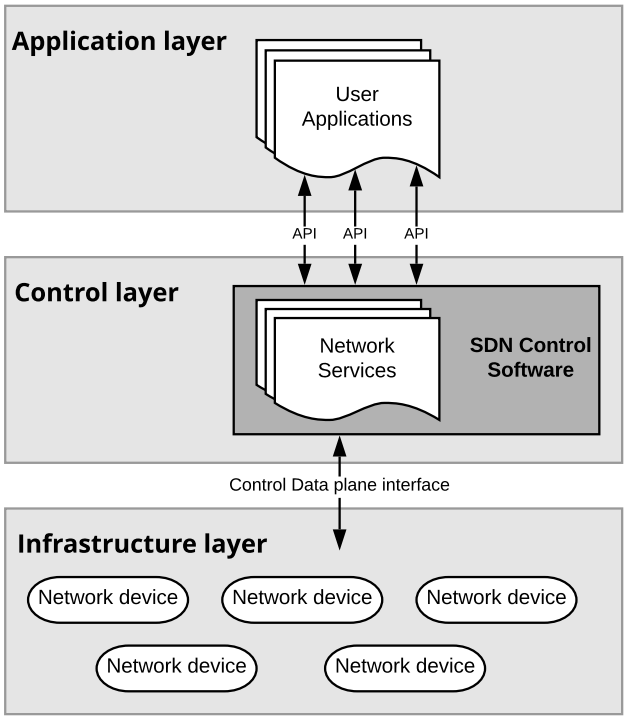

Figure 2.1 SDN system architecture

2.2.2 SDN advantages over traditional network

The virtual architecture of SDN provides quite a few novel functionalities over traditional network architectures:

- It enables the network administrators to easily and directly program the network controls. This is possible because of the decoupling of the data layer and the control layer, allowing swift dynamic traffic engineering and traffic-adjustment as needed.
- The SDN architecture allows a central network management by the controllers acting as the a logical vantage point and having a global view of the entire network. This allows an abstraction of the actual network from the applications that run the controller and the policy engine as a single logical switch.
- SDN enables dynamic configuration management of all the network devices through programmable SDN applications that run on the contoller. These applications are written using a set of APIs and protocols, such as OpenFlow, to control and manage the underlying network without worrying about any proprietary software.
- It also enables the network security administrators to have a global perspective of the whole network, making it easier for them to view all active flows froma single vantage point which is the controller. This allows the administrators to apply reactive measures based on the context, and deploy them instantly across the entire network.
- Since SDN is vendor-neutral and follows open standards, it provides the freedom and flexibility of incorporating its primitives in a wide range of hardware and also simplifies the network design.

2.2.3 OpenFlow standard

OpenFlow (OF) is considered one of the first software-defined networking (SDN) standards. It originally defined the communication protocol in SDN environments which enables the SDN controller to directly interact with the forwarding plane of network devices, such as switches and routers. It enables the network controllers to determine the path of network packets across a network of switches. These controllers are distinct from the OpenFlow switches, and thus the separation of control from the forwarding allows for more sophisticated traffic management than is feasible using traditional mechanisms like access control lists (ACLs) and routing protocols. Since OpenFlow is an open networking standard, it also allows switches from different vendors - often each with their own proprietary interfaces - to be managed remotely.

To work in an OpenFlow environment, any device that wants to communicate to an SDN controller must support the OpenFlow protocol. Using this interface, SDN controller pushes down changes to the switch or router flow-table allowing network administrators to partition traffic, control flows for optimal performance, and test new configurations and applications. For the network switches, there is an OpenFlow compliant soft switch call Open vSwitch which is widely used. On the controller side, several solutions exist most popular of which are OpenDaylight platform (Java) and the Ryu or Pox platforms (Python).

The OpenFlow standard allows manipulation to flow-table entries in an OpenFlow compatible switch from an OF controller in the following forms:

- Rule: This consists of a tuple of the source and destination addresses, ports, packet type, and VLAN ID numbers. Any of these fields can be used to create a rule on which a packet is matched against.

- Action: The action field tells the switch what to do when a packet is received which matches the rule defined above. Some of the basic actions that can be performed by a SDN switch include: forwarding the packet to a specific port, encapsulating and forwarding it to the controller, dropping the packet, or send the packet through the normal processing pipeline.

- Stats: An OF switch also provides the controller with a high level abstraction of the networking hardware and thus this field provides information on the packet and the byte counters for a given flow table.

Because of all these functionalities, OpenFlow standards provide a lot of benefits for the virtualization of networks, including:

- Programmability - It enables innovation of networking applications, and accelerates new features and services introduction.

- Centralized Intelligence - It simplifies the provisioning of networking services, optimizes performance, and allows for a granular policy management.

- Abstraction - By decoupling of hardware & software, control plane & forwarding plane, and physical & logical configurations, using OpenFlow standards provide a much richer abstraction of the network than traditional systems.

In this work, we seek to introduce the benefits of a SDN based network deployment in a smart home in order to achieve a granular control over the communication between smart home devices on the local network.

CHAPTER

3

RELATED WORK

Our work touches several areas of prior research, which we have divided into three broad categories. First, we re-iterate our threat model for smart homes by discussing studies on IoT device compromises and the security of smart home platforms. Next, we discuss the current measures being taken by the industry and security researchers in providing a defense mechanism for smart homes. Finally, we present access control mechanisms proposed for IoT devices and provide insights on the need of a network based access control for smart homes.

3.1 Internet of (vulnerable) Things

Home automation and connected home environments have become an irresistible trend for both manufacturers and consumers. However, more often than not, features tend to take priority over security in these devices. Several security flaws have been identified by researchers in these off-the-shelf devices for smart home. It has been shown that some of these devices mistrust other devices on the same LAN, thus leaving them vulnerable to a local adversary [Pau13; Fer16b]. Several incidents of device compromises include light bulbs performing DoS attacks [Hil15b], attackers unlocking smart door locks using ZWave protocol vulnerabilities [FG13], and directed attacks on baby monitors from the internet [SB15; NBC14; Smi13]. Denning et al. [Den13] presented a taxonomy of IoT attacks including threats to smart home devices like cameras, microphones, motion sensors, door locks, HVAC air pressure sensors, thermostats, washing machines, and health care technologies. It seems like most of these devices tend to suffer due to control signals emerging from unauthenticated sources. For instance, researchers have found that an adversary can control a smart TV with a speaker playing synthetic voice commands [MK14]. Ho et al. [Ho16] demonstrate an attack on Bluetooth smart lock, similar to the attack on ZWave smart lock [FG13], where improper trust allows attackers to open locked doors. Additionally, implementation flaws by users can also lead to severe attacks against smart homes as demonstrated by Oluwafemi et al. [Olu13], where the compromise of a networked device can even affect non-networked devices such as compact fluorescent light bulbs. HoMonit [Zha18] compared the SmartApps activities as infered from side-channel analysis of encrypted wireless traffic against their expected behaviour as dictated in their source code, and found 60 misbehaving apps which conducted event-spoofing attacks.

The threat model for our implementation is based on these attack scenarios where the

compromise of one connected device may affect other devices in the same network. Our work addresses these issues with smart home devices by providing a default policy of network isolation that can mitigate device compromises in a smart home. Consequently, we do not intend to prevent users from connecting potentially vulnerable devices to their network but instead provide a scalable and effective measure of isolating network communications between these IoT devices, which in turn would prevent device compromises in the local network.

3.2 Smart Home defenses

Several different methods have been proposed to defend the communication between devices in a smart home. We discuss these methods in the following categories.

3.2.1 Authentication Schemes

In order to control communications between IoT devices, authentication schemes [HR15] have been tailored for resource constrained devices. Zenger et al. [Zen16] proposed a vicinity based pairing mechanism which delegates trust from one node to another based on physical proximity. However, these mechanisms require additional implementation on all devices of the system and do not consider the trivial problem of the existence of legacy devices in the network.

3.2.2 Behavioral security measures

Fully automated techniques have also been proposed to identify malicious communications within the smart home network. These mechanisms depend on intrusion detection systems

tailored for IoT scenarios [Raz13]. IoT Sentinel [Mie17] is such a system that identifies the types of devices connected to an IoT network and enable autonomous enforcement of rule-based-control policies. But the main limitation of their work is that they enforce these policies in three pre-determined isolation levels, *strict*, *restricted*, and *trusted*. This greatly affects the scalability of their system for increasing number of smart home device functionality. IoTSense [Bez18] presents another such behavioral fingerprinting method which can then be used to train a behavioral classifier to identify abnormal behavior in devices as demonstrated in IoTurva [Haf17]. But a drawback in using such de-centralized behaviour based policy generation is that the ground truth for behavior classification is obtained through crowd sourcing or by analyzing the CVE database. This methodology is not really a dependable measure for users, and will in more cases than not cause limitations in the device connectivity in smart homes.

3.2.3 Network-level security

This security measure is routinely used in enterprise networks as an addition to protection software installed in clients. Considering the heterogeneity of devices in smart homes, network-level security is even more relevant. The privacy control mechanism proposed in [Siv15] uses SDN-enabled switches of the ISP to dynamically block/quarantine devices, based on their network activity and on the context within the house such as time-of-day or occupancy level. However their implementation assumes that ISPs have complete visibility of the subscribers household devices, which raises a lot of privacy concerns as discussed in [Apt17]. Preliminary prototype envisioned for IoTSec [Yu15] demonstrates various ideas of a network based abstraction for IoT devices and shows its promises in implementation by providing security gateways for insecure devices and by enforcing policies for cross-

device interactions. Thereby reiterating the importance of achieving an efficient network abstraction in smart homes. Their design principles are the most similar to our approach in concept however, our network abstraction provides a much more simplistic architecture and doesn't have the trust issues associated with crowd sourced repositories for policy enforcement. Recently certain commercial products have also been announced which address the security of IoT devices in a smart home. One such product is the Sense router from F-Secure [FS18a]. However, it is only focused on traditional protection technologies like anti-virus capabilities, blocking of botnet controllers and preventing devices from accessing malicious websites [FS18b].

3.2.4 IDS and Firewall

Several works on intrusion detection systems (IDS), personal and application firewalls [App99; Che03; Cro07; JA02] focus either solely at the host or at a network node. IDIoT [Bar18] claims to provide a network security policy enforcement framework for IoT devices, but their design only creates policies on the gateway system monitoring device to cloud endpoint communications. Additionally, these traditional mechanisms remain oblivious to device-to-device communications in wireless networks.

The policy design in *Hestia* aims to provide device-to-device granular policies like IoT-Sentinel but doesn't depend on any third party to make these decisions. Hestia provides the smart home owners with a scalable and easy to configure policy framework to employ a least privilege policy in their home network.

3.3 Access Control in Smart Homes

Ur et al. [Ur13], in their study of access control in commercial smart devices, found that all these devices lack the mechanisms of access monitoring therefore making users incapable of identifying who has accessed their devices. This issue has been partially addressed with user-based access control policies [Kim10; He18] which envision a policy enforcement for different types of network users (owners, spouses, maids, neighbours). Some recent work more relevant to our design, which provide device specific access control are mostly either a permission-based control on appified platforms, or situation based control.

3.3.1 Permission-based access control

Since smart home platforms are accompanied by appified platforms, permission-based access control has received a lot of attention by the security research community. In the meantime, researchers have ventured into identifying vulnerabilities in the emerging IoT frameworks that enable building apps to communicate with smart home devices and wearables [Sam18]. Evidently so, researchers have demonstrated [Siv16] how malicious apps running on a users phone could exploit the lack of security mechanism in UPnP protocol to scout home network for IoT devices and expose them at will to external attacks. To address these kind of exploits, researchers have focused on providing access control mechanisms for malicious apps running on a smart phone in a smart home. SmartAuth [Tia17] provides users with an authorization user interface to bridge the gap between the functionalities explained to the user and the operations the app actually performs. The security frameworks proposed in FlowFence [Fer16a] provide a sandbox mechanism to operate on sensitive data by forcing applications to declare their intended data flow patterns. HanGuard [Dem17] addresses this

issue through SDN-driven data protection by running a user-space monitor on the users phone that detects authorized application access to IoT devices. ContexIoT [Jia17] provides a context-based permission system for appified IoT platforms, with access control at the inter-procedure and data flow levels. While these approaches promise enhanced security for accessing IoT devices, a key observation lies in their adversary model which is focused on the situation where a malicious app is installed on a smart phone device. However, as covered in the earlier section on IoT vulnerabilities, we need to consider unpatched devices with vulnerable firmwares in our adversary model as well.

3.3.2 Situational access control

Few works have addressed access control in IoT based on a situation - like "user is at home" - that can only be tracked using multiple devices. Schuster et al. [Sch18] provide a system which uses environmental situation oracles (ESOs) which encapsulates how a situation is sensed, inferred, or actuated. This methodology has been widely studied for Android platforms and other mobile OSes [Bug13; Con10; Nau10; Wij15]. But the critical distinction between IoT and mobile OSes is that access control in the IoT is fundamentally decentralized. While ESOs seems like a promising way of incorporating access control in the IoT stack, it needs a configuration change on all existing devices and automation systems. It might be an access control that we deserve, but not the one we need right now.

Closest to Hestia, prior work has provided policy for device communication [Bar18] and monitored device connections using gateway controller devices [Mie17; Yu15; Haf17]. Barrera et al. [Bar18] address device-to-cloud communication by whitelisting device connections at the gateway router. However, their fine-grained policy will be difficult to manage as devices gain features allowing customization or extensibility. Since certain smart home

products support interfacing with a variety of cloud services, enumerating all possible features in an exhaustive firewall policy will become infeasible. To address the security of device-to-device communication, IoTSec [Yu15] analyzes data flows from IoT devices, and allow interactions based on an application level policy. IoT-Sentinel [Mie17] and IoTurva [Haf17] are designed with a similar architecture in mind. They create ad hoc network overlays for connected devices and use OpenFlow rules to constrain the communication from vulnerable devices. They provide sophisticated ways of identifying a vulnerable device (through an IoT Security Service) and confine its traffic flows using network isolation policies. Their enforcement mechanism, however, only addresses unicast communication from devices. This means that an adversary could still use multicast packets to gain sensitive information from connected devices.

Unlike prior work, Hestia achieves enhanced isolation between devices by mediating service discovery and providing default network policies that are validated for effective deployment. Hestia provides a default access control mechanism for smart home devices, approaching least-privilege, being user-configurable to scale with the changing smart home environment, and simple enough to be readily deployable today.

CHAPTER

4

MOTIVATION

In this work we are interested in applying a least privilege network policy to reduce the risks posed by vulnerable IoT devices to other devices on the local network. The critical challenge in a least privilege policy is: how do we determine what privileges a device needs? While industry mechanisms like MUD specification [Lea18] can provide fine grained device behavior descriptions, they are currently not implemented for today's devices, and may never be implemented for low-cost offerings. Other proposed systems [Mie17; Bez18; Haf17] leverage machine learning to learn device behaviors, but still ultimately result in complex, unwieldy policies.

We hypothesized a simpler solution could be equally effective. We begin with the hy-

pothesis that IoT devices primarily connect only to a smart hub or *controllers*, and not to other IoT devices or user devices (e.g., laptops). Loosely, controller devices receive control inputs from users, like Google Home or Alexa hubs, and execute certain actions on other non-controller IoT devices, like door locks or light bulbs. If this hypothesis held, we realized that a least privilege policy need only allow data flows between individual IoT devices and controllers, preventing compromised IoT devices from successfully targeting other non-controller devices. Such a policy can be implemented simply, with far less complexity or user input than other methods. In this section, we demonstrate using a previously-published large IoT dataset [Alr19] that this approach is feasible, leading the way to the development of our system, *Hestia*. The following section then provides an insight on the goals of such a design followed by an overview of our system.

4.1 Policy Evaluation

In prior work, Alrawi et al. [Alr19] released the YourThings dataset that includes 46 labelled smart home devices and the network traffic they generated during a manual assessment of each device. They recorded the device interactions on the network at 5 minute intervals over a period of 10 days. We use these captures to validate our hypothesis that a mere distinction of controllers and non-controllers is sufficient to achieve least privilege network policy.

4.1.1 Experiment

To test our hypothesis, we first manually classified the smart home devices in the YourThings data set into controllers and non-controllers. Our classification criteria was that devices which have one or more user-facing control interface (voice, app, etc.) and can execute

actions on other IoT devices are controllers, while any other IoT devices are not. For instance, if a smart TV has Alexa integrated and the manufacturer's website details that it can be used to control other devices in the network then we identify it to be a controller. Whereas a smart light bulb may be accompanied by an app on the users smart phone, but in no scenario can it send instructions to any other IoT device in the network. In most cases, necessary information about the device was accessible directly from the manufacturers landing page for their device. This process was fast and could be repeated by users with limited technical expertise.

Table 4.1 shows the number of devices that were classified as controllers and non-controllers in the data set and the distinguishing features used to determine this classification. We identified a total of 20 out of the 46 labelled devices to be a controller in the network.

The next step was to filter the recorded network data for local network communication and evaluate our policy. We used the `scapy` Python library to filter all the local network packets from each network capture and aggregated all device communications for each day of the recordings. Once we had the aggregated network communication data, we created a *src - dst* mapping of devices where a unique mapping was created on a per-day basis.[1] The goal was to achieve an insight on which devices are able to send messages to each other (within each 24-hr period), and thus if we found at least one instance of a packet exchange between two sets of devices on the network then we recorded it in our mapping.

4.1.2 Findings

Using the *src - dst* mapping of device interactions obtained for each day, we found that there were a total of 426 device-to-device communications that took place over the 10 day period.

[1]The devices and configurations vary from day-to-day in the YouThings dataset

Table 4.1 Device categorization on the YourThings data set

Distinguishing Feature	No. of Devices	Category
None	26	non-controllers
Voice Assistant	10	controllers
Remote Control Hub	9	controllers
Home Router	1	controllers

Since we created these mappings on a day-to-day basis, these include repeated instances of device-to-device interactions over several days. We determined that 54.69% of these device interactions were between devices that we had identified to be acting as the controllers in the network. We also found that 45.07% of connections occurred between controllers and non-controllers preserving our hypothesis that non-controllers need controllers to function. All of these packets conformed to our initial hypothesis that communication from IoT devices is constrained to controllers within the local network.

Our analysis revealed a single exception to the policy. We found that two packets were exchanged between two non-controllers: a D-Link DCS5009L camera and a Belkin Netcam. This was a UPnP device discovery initiated by the D-Link camera requesting the device details of Belkin Netcam. This device discovery was spurious and was not necessary for legitimate functionality. In fact, it serves as an example of the unnecessary and unauthorized network traffic we seek to prevent. While we believe this traffic was innocuous, it is similar to known attacks against the Belkin Netcam [phi18].

4.1.3 Takeaway

These empirical results validate our hypothesis that a simple network policy limiting IoT device traffic on the local network to only controller devices can be deployed in practice today with virtually no disruption to the proper function of these devices.

4.2 Problem

As demonstrated in Section 4, non-controller devices only require network communication with the Internet and controller devices. This observation offers an opportunity to enforce a network access control policy that is both simple to specify and approaches least privilege. In this section, we describe the problem via an example scenario. We then define the threat model for Hestia.

Problem Scenario: Setting up a seemingly innocent user appliance, like a smart coffee maker, involves three key steps. First, the user installs the coffee maker in the desired location and uses an accompanying smartphone app to connect to it, often via Bluetooth. Second, the coffee maker attaches to the home WiFi network. It either automatically uses the same WiFi SSID as the smartphone, or discovers all available WiFi SSIDs and asks the user, through the accompanying app, to select one and provide credentials. Finally, as the coffee maker connects to the WiFi network, remaining connected until changes are made by the user. Note that users are sometimes recommended to use network segmentation, or a separate SSID just for smart home devices; however, doing so complicates setup (e.g., if the SSID of the smartphone is cloned) and often adds network management overhead for average users.

The purpose of a smart coffee maker is quite specific: it brews coffee based on a schedule or when it receives a command from the user. It needs to be connected to the home WiFi network, and receive these user commands only from a supported controller platform. The coffee maker is a non-controller device and thus should not communicate with other non-controllers, like the printer, the refrigerator, or the security camera on the network. By having full access to the LAN, the coffee maker is thus over-privileged. Because these

smart home appliances are essentially fully-featured, Internet-connected Linux computers, vulnerabilities in their implementation [Dic19] mean attackers can use it as a pivot to attack the entire network. This paper seeks to define an approximation of least privilege network access for smart home devices that would mitigate such a network compromise, while balancing usability.

Threat Model: We consider a smart home network where users have multiple smart home appliances, like a smart coffee maker, all connected to a consumer-grade router. By deploying all devices on a shared network space, users put the same level of trust on all connected devices. We assume that an adversary can discover the vulnerable devices in a users' household using malicious smartphone apps [Siv16], by exploiting HTML error messages [Aca18], or by simply gaining physical control of the device [Dha15]. Once the attacker has gained access to a device, they can maliciously gather sensitive information from other connected devices (e.g., unique device identifiers) [Aca18], use lateral movement to steal user data from the network-attached storage unit [Wei18], or discover a vulnerable security camera on the local network and attack it remotely [Gre17]. Network compromises such as these are known as "Rube Goldberg" attacks, where attackers penetrate the local network through a vulnerable device, and then use it to exploit other connected systems. Note that many user devices, including NAS, coffee makers, and security cameras, are non-controller devices and do not need to interact with each other. Like other network access control systems (e.g. firewalls), we recognize that detection of attacks (e.g. NAT hole punching) and compromised devices (e.g. botnet infections [Ant17]) will have orthogonal solutions [FS18b; Mie17].

CHAPTER

5

HESTIA

As motivated by our empirical evaluation in Chapter 4, Hestia is designed to enforce a simple policy: *non-controller devices may only communicate with controllers and the WAN*. This allows for a simple policy specification, only requiring knowledge of the link-layer addresses of controller devices. While not explicitly included or evaluated in our design, we reasonably assume the existence of a mechanism for an end user to designate a device as a controller, either *a priori* or during the first connection. Enforcing this simple policy requires overcoming the following research challenges.

- *Existing LAN network access control mechanisms do not mediate between devices on the*

same LAN. VLANs and separate WiFi SSIDs unnecessarily complicate network setup, and are incompatible with many consumer smart home devices that (a) automatically use the SSID of the smartphone performing setup, and (b) use multicast discovery protocols.

- *Multicast discovery packets from non-controller devices must not reach other non-controller devices*. Discovery protocols such as mDNS and SSDP commonly use multicast packets. Network layer multicast packets are broadcast to the link layer, allowing devices to respond if they are configured for that multicast address.

Hestia addresses the first research challenge using Software Defined Networking (SDN) primitives. Specifically, we modify an OpenWRT firmware to include an OvS (Open vSwitch) soft switch that relays packets to a Ryu-based SDN controller application (potentially running on the router). Through this SDN-based instrumentation, Hestia is able to mediate network communication between devices on the same LAN, without the need for VLANs or multiple SSIDs. To address the second challenge, Hestia provides a generic solution that supports mDNS, SSDP, and other multicast discovery protocols without specific knowledge of the protocols. To do so, Hestia uses the group table feature of OpenFlow 1.3 to selectively duplicate multicast packets at the link-layer and send a unicast version to controller devices.

Figure 5.1 overviews the Hestia architecture. The WiFi router instrumented with Hestia is configured with the MAC addresses of all controller devices. By default, a device is a non-controller device. Since non-controller devices can communicate with controller devices and the WAN, most devices will work automatically when they are connected to the LAN. When the first packet of a new flow arrives at the router's WiFi interface, the Open vSwitch sends the packet header information to the Ryu-based SDN controller application. The SDN controller inspects the source and destination MAC addresses. If the flow is allowed,

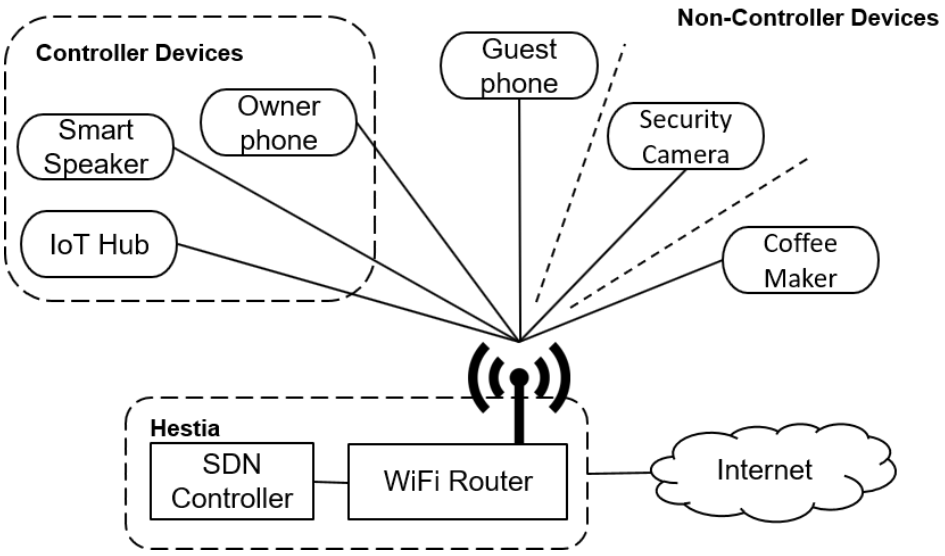

Figure 5.1 Virtual device partitions created by Hestia in a smart home

an OpenFlow flow-mod is sent back to Open vSwitch, permitting subsequent packets in the flow to forward without the involvement of the SDN controller. If the destination MAC address is one of a set of multicast addresses used by device discovery protocols, Hestia uses the group table feature of OpenFlow 1.3 to define action buckets that automatically duplicate the payload and unicast send it to controller devices. The remainder of this chapter describes the various aspects of Hestia's design.

5.1 Enforcement

Hestia is configured with a list of identified controller devices. Our implementation uses a JSON file enumerating the MAC addresses. We envision two ways for Hestia to obtain this list. A straightforward way is for the user to identify each controller device when it is added to the WiFi network for the first time. This specification could occur via the router's configuration web interface, or via a companion smartphone application. As an automated

alternative, Hestia could be combined with IoT device fingerprinting system such as IoT Sentinel [Mie17] (though "controller" annotations would be needed). Even with automation, Hestia should allow the end users to manually designate controller devices as needed (e.g., to add specific non-IoT devices such as smartphones and laptops). Note that not every smartphone or laptop needs to be a controller device. Only those requiring connections to other devices on the LAN need to be controllers.

Hestia uses Open vSwitch to mediate packets between all devices on the LAN. In normal WiFi access points (APs), the wireless interface behaves as a bridge between the wireless clients. If a frame originating from a wireless client is destined for another client connected to the same wireless interface, then the frame is directly sent to the other client without going through the network stack (bypassing Open vSwitch). To force frames to go through the network stack, we enable wireless isolation mode (also called AP isolation or peer-to-peer blocking), which is available in standard APs. By default, wireless isolation will prevent all wireless clients from communicating with one another. However, SDN can selectively re-enable communication that adheres to the policy [Hat16]. To do so, we configure Open vSwitch to forward packets to the ingress port (`OFPActionOutput(ofproto.OFPP_IN_PORT)`) when communication is between two wireless clients.

We implemented the Hestia SDN controller application using Ryu. Ryu is relatively lightweight and Python-based, allowing it to run on newer commodity routers with more resources, or via a directly connected device (e.g., Raspberry Pi). Hestia installs flow rules based on the device categorization to create a categorical entity abstraction, as shown in Figure 5.1. Devices are classified as "non-controller" by default. All non-controller devices are isolated such that they can initiate connections to the internet and controller devices, but not to other non-controller devices. Therefore, a controller device not designated as such will still have partial functionality (e.g., internet connectivity). Once a device is

specified as a controller, it may act as a traditional device on a LAN. Finally, whenever the controller list is updated, Hestia flushes all OpenFlow rules to ensure proper operation. In effect, this enforcement maps to a loose approximation of least privilege based on the network functionality, as we observed in Chapter 4.

5.2 Selective Device Discovery

Since most homes use dynamic IP assignment (i.e., DHCP) and do not run internal DNS servers, controller devices use discovery protocols to connect to smart home devices. For example, mDNS and SSDP discover devices by sending multicast packets, which reach the NIC of all devices on the LAN. Devices configured as mDNS or SSDP listeners consume the multicast packet and respond using protocol-specific conventions. Hestia carefully controls device discovery packets to prevent non-controller devices from performing network reconnaissance and discovery protocol-based attacks.

Hestia seeks a generic approach for handling device discovery that works without protocol-specific knowledge of mDNS and SSDP. Standard network implementations handle multicast packets by copying the original packet from the source and broadcasting it to all connected devices with a MAC layer destination address of the desired multicast group. To achieve selective isolation for multicast communication, Hestia must control which devices can receive the multicast packet.

To efficiently mediate multicast device discovery protocols, Hestia uses the *group table* feature in OpenFlow 1.3. Group tables allow an SDN controller to define action buckets that automatically duplicate the packet for each action target. When a non-controller sends a multicast packet, Hestia creates a group table that matches on the source MAC address of the non-controller and the destination MAC address of each multicast group. We then

use the action buckets to specify an action for each controller MAC address. In doing so, Open vSwitch effectively converts multicast packets into unicast packets, without sending the packet payload to the SDN controller application. Note that multicast packets from controller devices are forwarded as normal.

5.3 Implementation

We implemented Hestia on top of a custom OpenWrt firmware with Open vSwitch and Hostapd pre-compiled. We removed the default Linux bridge and separately added an Open vSwitch bridge. We used the Ryu SDN framework and implemented the Hestia SDN controller in 320 lines of Python. In this section, we discuss the policy generation algorithm, and how we incorporate the various OpenFlow components to support our design.

5.3.1 Policy generation algorithm

Hestia affects every device connection that traverses through the router. Our design specifically monitors packets that need to be routed on the local network, and creates the required data flows to handle these cases. We implemented a SDN app in RYU, called HestiaSwitch, which logs each packet_in event on the open vswitch and retrieves the tuple (src, dst, in_port) from the layer 2 packet headers. Using the information obtained from this tuple, accompanied by the device categorization obtained from the user, HestiaSwitch creates the requisite data flows to approach a least privilege network access. In Figure 5.2, we show a high level understanding of our implementation starting with a packet_in event logged by the switch when a device sends a local network packet.

After obtaining the packet headers from the incoming packet, the algorithm checks the purpose of the packet to identify if it is meant to be routed on the local network or to

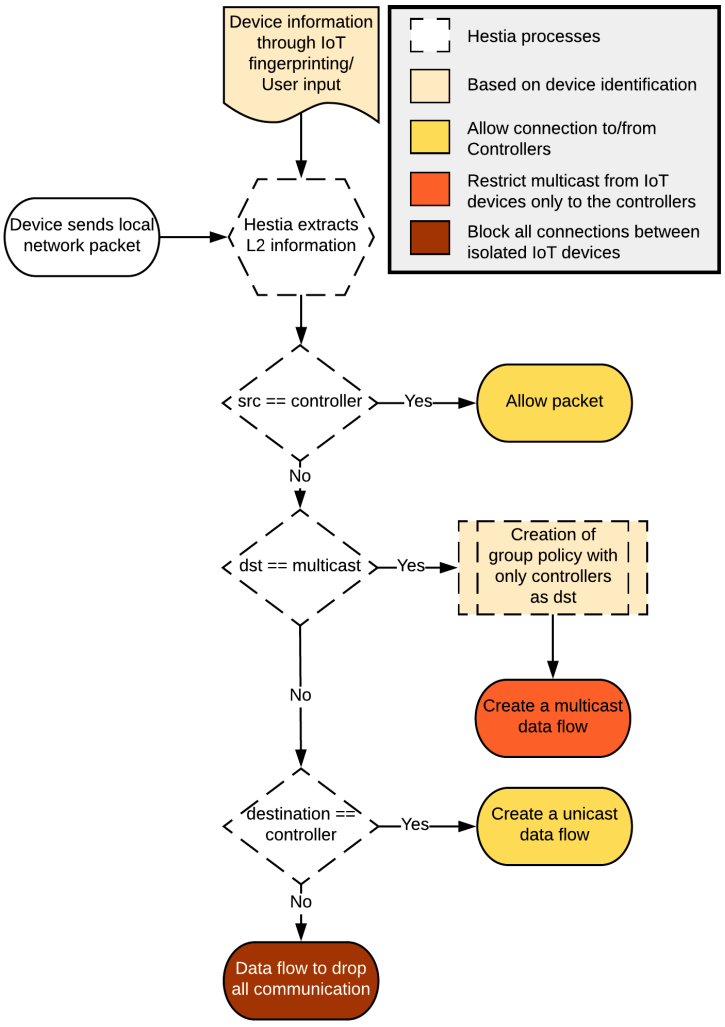

Figure 5.2 Policy generation algorithm of Hestia

the WAN. For the latter case, a default data flow is installed on the router to direct all such packets from that specific device to the WAN port. If the packet bears a destination address in the local network, we validate the flow against the device categorization and create the required policy.

5.3.2 Device Categorization

The device categorization details, given as an input to Hestia, is in the form of a json object as shown in listing 5.1. We store the MAC addresses of all the controller devices connected to the access point in a database. This enables our design to function as a plug and play architecture that can be integrated with IoT device fingerprinting systems like IoT-Sentinel [Mie17] or with smart phone apps for home routers [Tec18; ASU18; Ser18]. This presents the convenience for users to load their desired configuration at the time of deployment of their device in their smart home.

Listing 5.1 JSON object providing a list of controllers

```
{ "controllers": ["00:aa:bb:cc:dd:ee","a1:11:bb:22:cc:33",
                  "a2:23:cc:dd:33:44"] }
```

In our implementation, we fetch this json object from a database that the user has access to. The idea here is to allow the user to push updates to the configuration at their convenience, either through a web based interface or through a smart phone app. It is also important, that with every update to the data store, the installed network policies are also updated. For instance, if the user accidentally adds a device to the controllers list and later decides to remove it, then an update to the configuration should reflect the same. Any non-controller device should not be able to maintain its (wrongly) privileged access on the network. In order to ensure this, we take two specific actions in the design of Hestia. The first step is to have the installed flows time out after a certain period and be re-installed by Hestia reactively. We achieve this by writing an idle timeout value to each flow. This ensures that any inactive device on the network has to re-negotiate with Hestia to gain its' required privilege. We also use a hard timeout value on each flow to have all connected re-negotiate their privilege with Hestia periodically.

We also implemented another SDN app which proactively clears all the installed flows on the switch if an update has been made to the configuration. This app runs in parallel to Hestia, and its' sole function is to managing updates to configuration. However, it is important that both the installed flows and group tables get updatd as we cannot have new flows pointing to outdated group tables. We achieve this by clearing out all the installed flows, and deleting all the residing group tables in the memory of the switch.

Listing 5.2 OpenFlow message to remove installed flows

```
{ "OFPFlowMod": {
      "command": OFPFC_DELETE_STRICT,
      "cookie": 0,
      "cookie_mask": 0,
      "flags": OFPFF_SEND_FLOW_REM,
      "instructions": {},
      "match": {},
      "out_group": OFPG_ANY,
      "out_port": OFPP_ANY,
      "priority": 123,
      "table_id": 0
} }
```

Listing 5.2 shows the openflow message that is sent to the open vswitch in order to remove the installed flows from the switch. We use the `OFPFC_DELETE_STRICT` command to achieve this as we only intend to remove the flows that correspond to connected devices. We still want to retain the flow for the SDN controller, which is a dedicated device in our implementation but can be integrated with the router as well. The `OFPFC_DELETE_STRICT` only deletes the flows that are of the same priority level (123), whereas `OFPFC_DELETE` will delete all installed flows including the one for the SDN controller. The flow installed for the SDN controller is usually assigned a priority level 0 (highest) and should not be deleted as it will inhibit proper functioning.

5.3.3 Generating policies

The very first validation that our algorithm performs is to check whether the source device is acting as a controller in the network. If this is the case then, by virtue of its privilege, any communication from that device should be allowed by default.

If however the source is not defined in the list of the controller devices, then it assumes the role of a non-controller. As we described in section 5.2, that in order to obtain an effective least privilege policy we need to implement selective duplication of multicast packets sent by a non-controller. So we need to explicitly distinguish multicast communication from unicast communication. In order to do that, we encode a list of multicast destinations that are commonly used by IoT devices to discover and advertise their services. This list essentially acts as the filter to create separate policies for multicast communication.

5.3.4 Group tables for multicast

To effectively handle multicast communication, we create a group table for that source with destinations limited to the controllers only. A group table consists of group entries, or an ordered list of action buckets, where each entry specifies a set of openflow actions to execute. So for every single packet from a non-controller device, the open vswitch duplicates the packet to match the number of group table entries and performs the actions that are specified in each bucket. An example of a group table entry converting a multicast packet to a set of two unicast packets for two controller destinations is shown in listing 5.3.

Listing 5.3 OpenFlow message to create action buckets for group flows

```
{ "OFPGroupMod": { "buckets":
       [ { "OFPBucket": { "actions": [
                      { "OFPActionSetField":
                             {"eth_dst": 01:aa:02:bb:03:cc} }
                      { "OFPActionOutput":
                             { "port": OFPP_IN_PORT}
                      }], }}
    "group_id": 1
}}
```

It can be noted that each OFPBucket uniquely defines the set of actions to be taken for each controller destination. For the case above, there are two controller devices whose MAC addresses are obtained through user categorization. Thus the group table created for any non-controller device will have the same group table containing two buckets. Each bucket has two OpenFlow actions that needs to performed by the open vswitch on each incoming mulicast packet. After creating this group table, a data flow is written to the switch with an action that matches the *(src, group_dst)* to the specific group table created for that flow. This ensures that all communication from that device to the specific group address always follows the required group actions.

5.3.5 Unicast flows

If the logged packet is not part of a multicast communication, then either of two situations are created. First situation is that it is a direct communication from a non-controller device to a controller device in the network. In this situation, the policy allows the communication to happen and thus writes a data flow on the switch supporting all future communications. The alternate situation is that the non-controller device is trying to directly communicate with another non-controller device in the network. We have earlier identified such commu-

nications to be spurious and unnecessary for the functioning of these devices. Thus a 'drop packet' data flow is written on the switch which avoids all future communications between those two devices.

Listing 5.4 OpenFlow message to install a FlowMod on the switch

```
{ "OFPFlowMod": {
      "hard_timeout": 20,
      "idle_timeout": 10,
      "instructions": [
         { "OFPInstructionActions": {
               "actions": [
                  { "OFPActionOutput": {
                        "max_len": 65535,
                        "port": 6,
                        "type": 0 }}],
            }}],
      "match": { "OFPMatch": {
            "oxm_fields": [
               { "OXMTlv": {
                     "field": "eth_dst",
                     "mask": null,
                     "value": "a1:a2:n1:m2:m3:cc"}}],
         }},
      "out_port": OFPP_IN_PORT,
      "priority": 123,
      "table_id": 1}
}
```

An example of an OpenFlow Mod is shown in listing 5.4, where open vswitch performs the actions specified if any new flow matches on the "eth_dst" field given. This format is used for both unicast and multicast communication, to install the requisite flows on the switch. The only difference in those two cases is the specification of the group table in the action field.

CHAPTER

6

EVALUATION

6.1 Network Performance

In the previous chapter, we described the design of Hestia. In this chapter we focus on evaluating how the deployment of Hestia affects the network performance of device communication. Since Hestia affects every communication that traverses through the home router, it is important that our model does not incur any significant overhead for the users' devices. In order to evaluate performance overhead, we designed experiments to measure latency and throughput for the communication between devices. Through our experiments, we show that Hestia has negligible performance impact.

6.1.1 Experimental setup

Our network evaluation explores a total 12 different experimental conditions. To evaluate the network performance of Hestia, we want to investigate the impacts to all communication types: device-to-device, device-to-cloud, and multicast messages. For all message types, we also consider the difference in non-controller-to-controller flows and controller-to-controller flows. For each measurement, we compare Hestia to the stock OpenWRT firmware and the default "Simple Switch" Ryu app.

We measure three key variables: first packet latency, average (non-first) packet latency, and average throughput. We distinguish the first packet of each flow from subsequent packets because SDN systems must forward the first packet of a new flow to the SDN controller to make a routing decision, increasing the latency of establishing the flow. Subsequent packets are not referred to the SDN controller and are unaffected by this additional latency.

Our experiment testbed network consists of a standard home router (Linksys WRT 1900 ACSv2) running OpenWRT LEDE 17.01, a Linux desktop serving as the SDN controller, a Macbook Air (Mid 2009) used to generate test traffic, and seven other user devices including smart phones, tablets, laptops, and eBook readers. Because we are evaluating WiFi performance, actual device type will have a negligible effect on latency and throughput. Since Hestia categorizes all IoT devices as either controllers or non-controllers, we performed separate sets of experiments with the Macbook acting as a controller and as a non-controller. Additionally, we configured all of other 7 devices as the non-controllers in the network so multicast packets are sent to all connected devices by the AP, ensuring uniformity across all experiments. We note that we use the desktop as the SDN controller for convenient deployment; in so doing, we overestimate the network latencies due to the controller. Future deployments can integrate the controller directly into the access point to

further reduce latency.

6.1.2 Latency of communication

For each round of the latency experiment we send 10 ICMP ping messages to measure latency between two devices or between the device and the WAN (`google.com`). We run 100 rounds of this experiment for each experimental condition, with a 5 second delay between rounds. When testing an SDN app, we cleared all flow rules to ensure that each run reflects the latency of a new flow.

Most latency measurement tools, including `ping`, do not support multicast latency experiments. To evaluate multicast latency, we developed a tool using the Python socket library that measures the time to receive TCP SYN-ACK packets in response to a SYN packet sent to a multicast address. We conduct this test with 2 receiving laptops and one sending laptop. For each experimental condition we run 100 rounds, with a 5 second delay between each round, and clear all flow rules between each run.

6.1.3 Throughput measurements

For each round of throughput measurements we use the `iperf` tool to conduct a 10-second TCP bandwidth measurement; each round was repeated 20 times for each experimental condition, with a five second delay between experiments. Like the latency experiments, we cleared all installed flow rules between each run.

6.1.4 Results

First, we can observe in Figure 6.1 that in all communication settings and devices, average latency is largely constant, indicating that Hestia performs on par with both a stock

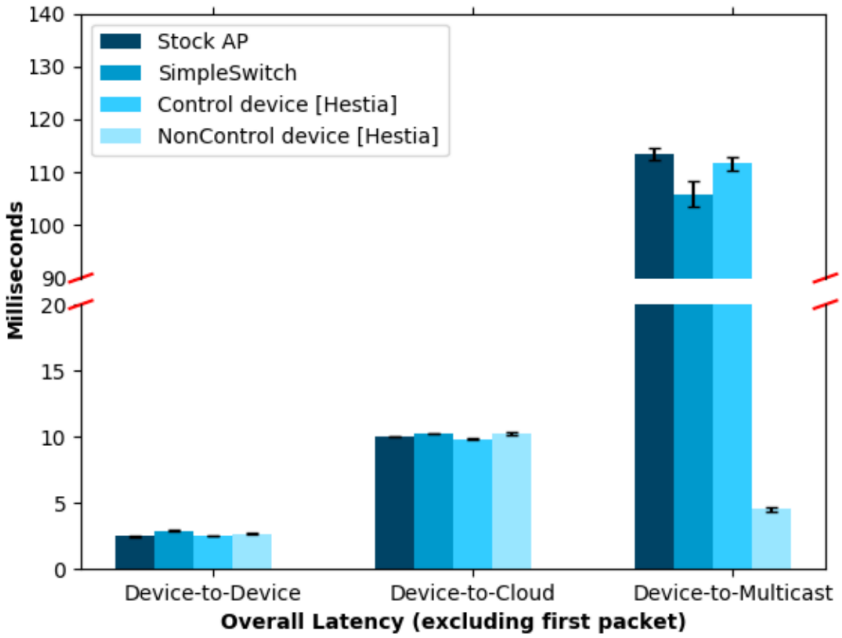

Figure 6.1 Latency of communication without the first packet

OpenWRT image and a stock SDN app. Naturally, we see that local communications have lower latency than those to the WAN. We also see that multicast packets have drastically higher latency; this is because multicast implementations wait a random amount of time before responding to avoid collisions. The one exception to these trends is in the non-controller multicast latency, which is lower than any other setting because the multicast packet is converted into a series of unicast packets and sent to only to a specific set of devices (controllers). This is also an indication that Hestia is providing isolation correctly.

Second, in Figure 6.2, we see that the latency for first packets only is significantly higher for all SDN systems, including both the SimpleSwitch app and Hestia, than for the stock access point. This is simply due to the first packet being forwarded to the controller for a flow decision. Hestia performs similarly to SimpleSwitch in all test cases except for non-controller multicast latency, which is faster for the same reason as the previous experiments.

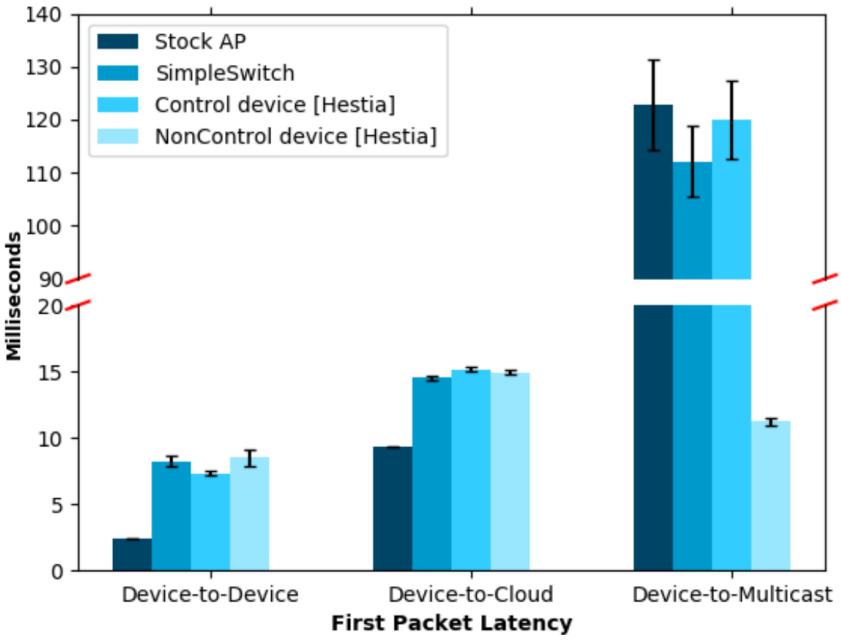

Figure 6.2 Measuring latency of only the first packet

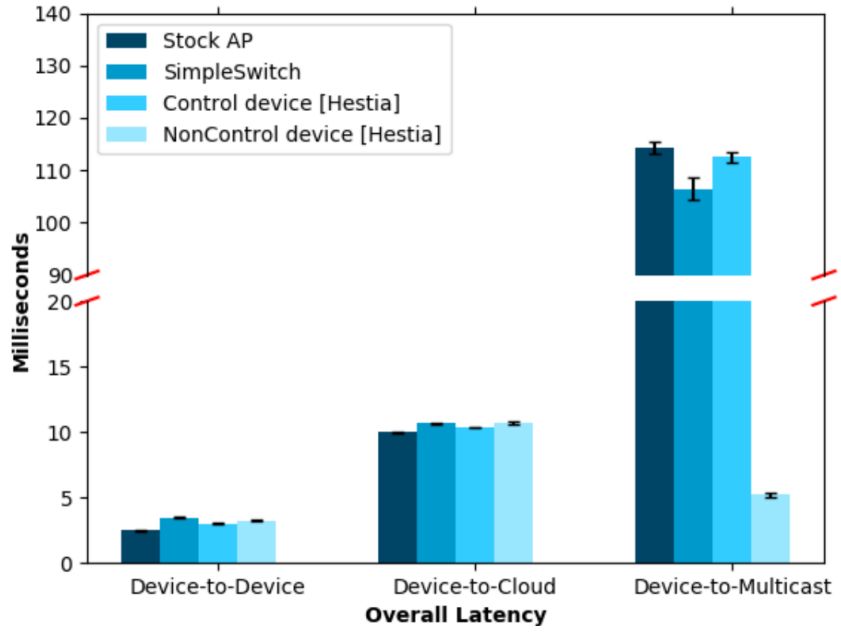

Figure 6.3 Overall latency of device communication

In figure 6.3 we show that the overall latency of device communication remains practically unaffected even after incurring a high first packet latency due to the implementation of SDN systems. This result bolsters the implementation of our model as an effective substitute for the standard home router, as a means to achieve least privilege network access for smart homes.

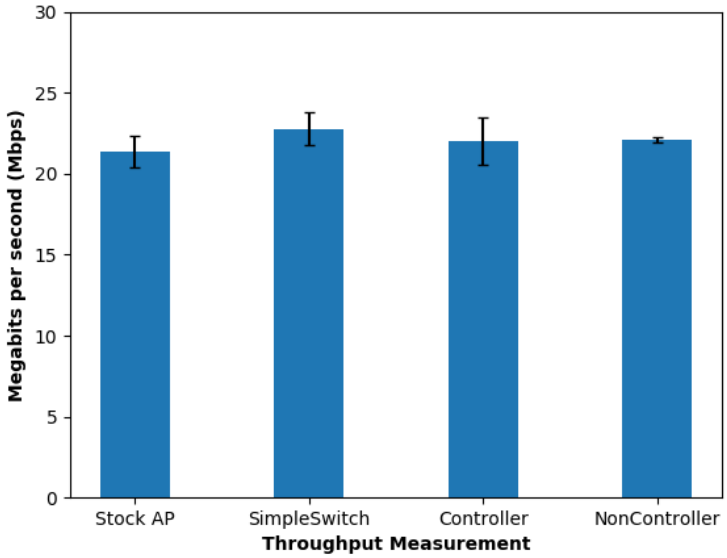

Figure 6.4 Comparison of measured throughput

Finally, Figure 6.4 shows the results of our throughput experiment. Similar to our previous experiments, we find that Hestia does not negatively impact throughput.

CHAPTER

7

DISCUSSION

Hestia creates least privilege policies of network access based on the identification of controller devices in a users home. We posit that this identification is trivial enough for the users to instinctively define during the initial setup of their smart home devices.

We identify only the controllers in the network and treat every other device as a non-controller. This allows Hestia to provide default internet access to all connected devices on setup, and only allow network visibility of other devices if the user specifies a device to require such functionality. This default mode of operation, however, does not guarantee complete functionality on setup as some devices (controllers) require network visibility to function. Thus, Hestia does not support complete functionality out of the box, but rather

depends on the user to provide the requisite configuration. We envision that the controller identification process can further be automated with advanced device identification systems like IoT-Sentinel, or through device type specifications like the Manufacturer Usage Description (MUD). We would also like to note that Hestia can be configured to act in conjugation with these systems, and not exclusively, thus allowing for more dynamic setups in a smart home.

The design of Hestia also places a trust on the home grade router, set up by the user in their smart home. We operate under the fair conditions that a user is well aware of their network setup, of the devices that are connected to their network, and have sufficient gateway security measures to prevent infiltration attacks to their local network. This assumption, however, does not reflect a complete security of the system. Attacks on the gateway router are fairly common in practice and are often attributed to the usage of default credentials by the user. By gaining access to the gateway router, an adversary can potentially alter the users configurations and assign a malicious non-controller device as the controller without the users' notice. This is a valid threat model for our system but is a more orthogonal concern to the research problem that we address in this work. However we envision that a possible mechanism of preventing this could be provided through a 2-factor authentication of configuration changes, where the user has to approve every change to their network setup through a 2FA code on their smart phone.

CHAPTER

8

CONCLUSION

The growing number of IoT devices emerging in consumers' home networks continues to raise significant security and privacy concerns. While efforts to improve IoT device security are important, the broad heterogeneity of devices and manufacturers necessitates network-based controls to mitigate the effects of compromised devices. Recently proposed network access controls for IoT devices provide great flexibility, but they require fine-grained policies that are difficult to specify. This work proposed a practical approach to the problem. We categorized devices into controllers and non-controllers and hypothesized that non-controllers only need to communicate with controllers and cloud servers. We validated this hypothesis using a public dataset of network traces of over 40 smart home devices. We then

provided Hestia, which enforces this intuitive policy. Hestia only requires users to specify which devices are controllers and introduces negligible performance overhead. As such, Hestia provides an effective and practical way to provide least-privilege access control in smart home networks.

BIBLIOGRAPHY

[Aca18] Acar, G. et al. “Web-based attacks to discover and control local IoT devices”. *Proceedings of the 2018 Workshop on IoT Security and Privacy*. ACM. 2018, pp. 29–35.

[Alr19] Alrawi, O. et al. “SoK: Security Evaluation of Home-Based IoT Deployments”. *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2019, p. 0.

[Ant17] Antonakakis, M. et al. “Understanding the Mirai Botnet”. *USENIX Security Symposium*. 2017.

[App99] Appenzeller, G. et al. “User-friendly access control for public network ports”. *INFOCOM’99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 2. IEEE. 1999, pp. 699–707.

[App18] Apple. *Apple HomeKit*. 2018. URL: https://www.apple.com/ios/home/.

[Apt17] Apthorpe, N. et al. “Spying on the smart home: Privacy attacks and defenses on encrypted iot traffic”. *arXiv preprint arXiv:1708.05044* (2017).

[ASU18] ASUS. *ASUS Router App*. 2018. URL: https://play.google.com/store/apps/details?id=com.asus.aihome&hl=en.

[Bar18] Barrera, D. et al. “Standardizing IoT Network Security Policy Enforcement”. *Workshop on Decentralized IoT Security and Standards (DISS)*. Vol. 2018. 2018, p. 6.

[Bez18] Bezawada, B. et al. “Behavioral Fingerprinting of IoT Devices”. *Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security*. ASHES ’18. Toronto, Canada: ACM, 2018, pp. 41–50.

[Bug13] Bugiel, S. et al. “Flexible and fine-grained mandatory access control on android for diverse security and privacy policies”. *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 2013, pp. 131–146.

[Che03] Cheswick, W. R. et al. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[Cim18] Cimpanu, C. *Over 65,000 Home Routers Are Proxying Bad Traffic for Botnets, APTs*. 2018. URL: https://www.bleepingcomputer.com/news/security/over-65-000-home-routers-are-proxying-bad-traffic-for-botnets-apts/.

[CNE15] CNET. *Google ties Android to the smart home with Brillo and Weave*. 2015. URL: https://www.cnet.com/news/google-announces-brillo-at-io-developers-conference/.

[Con10] Conti, M. et al. "Crepe: Context-related policy enforcement for android". *International Conference on Information Security*. Springer. 2010, pp. 331–345.

[Cro07] Crotti, M. et al. "Traffic classification through simple statistical fingerprinting". *ACM SIGCOMM Computer Communication Review* **37**.1 (2007), pp. 5–16.

[Dem17] Demetriou, S. et al. "HanGuard: SDN-driven Protection of Smart Home WiFi Devices from Malicious Mobile Apps". *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '17. Boston, Massachusetts: ACM, 2017, pp. 122–133.

[Den13] Denning, T. et al. "Computer Security and the Modern Home". *Commun. ACM* **56**.1 (2013), pp. 94–103.

[Dha15] Dhanjani, N. *Abusing the internet of things: Blackouts, freakouts, and stakeouts*. O'Reilly Media, Inc., 2015.

[Dic19] Dickson, B. *More Smart home devices vulnerable, McAfee researchers find*. 2019. URL: https://www.dailydot.com/debug/smart-home-mcafee-attacks.

[FS18a] F-Secure. *SENSE Security Router*. 2018. URL: https://www.f-secure.com/en_US/web/home_us/sense.

[FS18b] F-Secure. *What are the current projection features for f-secure sense?* 2018. URL: https://community.f-secure.com/t5/F-Secure-SENSE/What-are-the-current-security/ta-p/82972.

[Fer16a] Fernandes, E. et al. "FlowFence: Practical Data Protection for Emerging IoT Application Frameworks." *USENIX Security Symposium*. 2016, pp. 531–548.

[Fer16b] Fernandes, E. et al. "Security analysis of emerging smart home applications". *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 636–654.

[FG13] Fouladi, B. & Ghanoun, S. *Honey, I'm home!! - Hacking Z-Wave Home Automation Systems*. 2013. URL: http://www.blackhat.com/us-13/briefings.html#Fouladi.

[Fou18] Foundation, O. C. *AllJoyn Open Source Project*. 2018. URL: https://openconnectivity.org/developer/reference-implementation/alljoyn.

[Gre17] Greenberg, A. *Hack Brief: 'Devil's Ivy' vulnerability could afflict millions of IoT devices*. 2017. URL: https://www.wired.com/story/devils-ivy-iot-vulnerability/.

[Haf17] Hafeez, I. et al. "IOTURVA: Securing Device-to-Device (D2D) Communication in IoT Networks". *Proceedings of the 12th Workshop on Challenged Networks*. ACM. 2017, pp. 1–6.

[Hat16] Hatonen, S. et al. "Off-the-Shelf Software-defined Wi-Fi Networks". *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 609–610.

[He18] He, W. et al. "Rethinking access control and authentication for the home internet of things (iot)". *27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD*. 2018, pp. 255–272.

[HR15] Hernandez-Ramos, J. L. et al. "Toward a lightweight authentication and authorization framework for smart objects". *IEEE Journal on Selected Areas in Communications* **33**.4 (2015), pp. 690–702.

[Hil15a] Hill, J. *The smart home: a glossary guide for the perplexed*. 2015. URL: https://www.t3.com/features/the-smart-home-guide.

[Hil15b] Hill, K. *This guys light bulb performed a DDoS attack on his entire smart house*. 2015. URL: https://splinternews.com/this-guys-light-bulb-performed-a-dos-attack-on-his-enti-1793846000.

[Ho16] Ho, G. et al. "Smart Locks: Lessons for Securing Commodity Internet of Things Devices". *Proceedings of AsiaCCS*. 2016.

[Jia17] Jia, Y. J. et al. "ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms." *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. 2017.

[JA02] Judge, P. & Ammar, M. "Gothic: A group access control architecture for secure multicast and anycast". *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 3. IEEE. 2002, pp. 1547–1556.

[Kha12] Khan, R. et al. "Future Internet: The Internet of Things Architecture, Possible Applications and Key Challenges". *2012 10th International Conference on Frontiers of Information Technology*. 2012, pp. 257–260.

[Kim10] Kim, T. H.-J. et al. “Challenges in Access Right Assignment for Secure Home Networks.” *HotSec*. 2010.

[Lea18] Lear, E. et al. *Manufacturer Usage Description Specification*. Internet-Draft. IETF Secretariat, 2018.

[MK14] Michéle, B. & Karpow, A. “Demo: using malicious media files to compromise the security and privacy of smart TVs”. *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)* (2014), pp. 1–2.

[Mie17] Miettinen, M. et al. “IoT Sentinel: Automated device-type identification for security enforcement in IoT”. *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE. 2017, pp. 2177–2184.

[Nau10] Nauman, M. et al. “Apex: extending android permission model and enforcement with user-defined runtime constraints”. *Proceedings of the 5th ACM symposium on information, computer and communications security*. ACM. 2010, pp. 328–332.

[NBC14] NBCNews. *Man Hacks Monitor, Screams at Baby Girl*. 2014. URL: https://www.nbcnews.com/tech/security/man-hacks-monitor-screams-baby-girl-n91546.

[Olu13] Oluwafemi, T. et al. “Experimental Security Analyses of Non-Networked Compact Fluorescent Lamps: A Case Study of Home Automation Security”. *Proceedings of the LASER 2013 (LASER 2013)*. Arlington, VA: USENIX, 2013, pp. 13–24.

[Pau13] Paul. *Breaking and Entering: Hackers Say Smart Homes Are Easy Targets*. 2013. URL: https://securityledger.com/2013/07/breaking-and-entering-hackers-say-smart-homes-are-easy-targets/.

[phi18] phikshun. *Belkin Netcam HD UPnP Command Injection - Github*. 2018. URL: https://gist.github.com/phikshun/9984624.

[Raz13] Raza, S. et al. “SVELTE: Real-time intrusion detection in the Internet of Things”. *Ad hoc networks* **11**.8 (2013), pp. 2661–2674.

[Sam18] Samsung. *Samsung SmartThings*. 2018. URL: https://www.samsung.com/us/smart-home/smartthings/.

[Sch18] Schuster, R. et al. “Situational Access Control in the Internet of Things”. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 1056–1073.

[Ser18] Services, A. *Smart Home Manager*. 2018. URL: https://play.google.com/store/apps/details?id=com.att.shm&hl=en_US.

[SH13] Shekyan, S. & Hartutyunyan, A. "Watching the watchers: hacking wireless IP security cameras". *HITB* (2013).

[Siv15] Sivaraman, V. et al. "Network-level security and privacy control for smart-home IoT devices". *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 2015, pp. 163–167.

[Siv16] Sivaraman, V. et al. "Smart-phones attacking smart-homes". *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM. 2016, pp. 195–200.

[Smi13] Smith, M. *Eavesdropping made easy: Remote spying with WeMo Baby and an iPhone*. 2013. URL: https://www.csoonline.com/article/2225628/microsoft-subnet/eavesdropping-made-easy--remote-spying-with-wemo-baby-and-an-iphone.html.

[Smi14] Smith, M. *500,000 Belkin WeMo users could be hacked; CERT issues advisory*. 2014. URL: https://www.csoonline.com/article/2226371/microsoft-subnet/500-000-belkin-wemo-users-could-be-hacked--cert-issues-advisory.html.

[SB15] Stanislav, M. & Beardsley, T. *Hacking IoT: A Case Study on Baby Monitor Exposures and Vulnerabilities*. 2015. URL: https://www.rapid7.com/docs/Hacking-IoT-A-Case-Study-on-Baby-Monitor-Exposures-and-Vulnerabilities.pdf.

[Sta18] Statista. *Smart Home Market penetration in US*. 2018. URL: https://www.statista.com/outlook/279/109/smart-home/united-states.

[Tec18] Technologies, T.-L. *TP-Link Tether*. 2018. URL: https://play.google.com/store/apps/details?id=com.tplink.tether&hl=en.

[Tia17] Tian, Y. et al. "Smartauth: User-centered authorization for the internet of things". *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association. 2017, pp. 361–378.

[Ur13] Ur, B. et al. "The current state of access control for smart devices in homes". *Workshop on Home Usable Privacy and Security (HUPS)*. HUPS 2014. 2013.

[Ver18] Vera. *Vera, Smarter Home Control*. 2018. URL: https://getvera.com/controllers/vera3/.

[Waq17] Waqas. *How A Coffee Machine Infected Factory Computers with Ransomware*. 2017. URL: https://www.hackread.com/how-a-coffee-machine-infected-factory-computers-with-ransomware/.

[Wei18] Wei, W. *Casino Gets Hacked Through Its Internet-Connected Fish Tank Thermometer*. 2018. URL: https://thehackernews.com/2018/04/iot-hacking-thermometer.html.

[Wij15] Wijesekera, P. et al. "Android Permissions Remystified: A Field Study on Contextual Integrity." *USENIX Security Symposium*. 2015, pp. 499–514.

[Yaq17] Yaqoob, I. et al. "Internet of things architecture: Recent advances, taxonomy, requirements, and open challenges". *IEEE wireless communications* **24**.3 (2017), pp. 10–16.

[Yu15] Yu, T. et al. "Handling a Trillion (Unfixable) Flaws on a Billion Devices: Rethinking Network Security for the Internet-of-Things". *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. HotNets-XIV. Philadelphia, PA, USA: ACM, 2015, 5:1–5:7.

[Zen16] Zenger, C. T. et al. "Authenticated key establishment for low-resource devices exploiting correlated random channels". *Computer Networks* **109** (2016), pp. 105–123.

[Zha18] Zhang, W. et al. "HoMonit: Monitoring Smart Home Apps from Encrypted Traffic". *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 1074–1088.

APPENDIX

APPENDIX

A

HESTIA SOURCE CODE

The core part of Hestia is a SDN controller app created in RYU. The "HestiaSwitch" app is responsible for fetching the device configuration details from a specified medium and then enforcing the openflow policies on the soft switch on the wireless access point.

Listing A.1 Creating a RYU controller app

```
class HestiaSwitch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
```

Listing A.1 (Cont.): Creating a RYU controller app

```
    def __init__(self, *args, **kwargs):
        super(HestiaSwitch, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        config.configuration = config.update_config()
        self.controllers = config.configuration['controllers']
        self.filter = config.configuration['multicast']
        self.group_id = config.group_ctr


    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        #self.add_flow(datapath, 0, match, actions)
        inst = [parser.OFPInstructionActions(ofproto.
            OFPIT_APPLY_ACTIONS,
                                             actions)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=0,
                                match=match, instructions=inst)
        datapath.send_msg(mod)


    def add_flow(self, datapath, priority, match, actions, buffer_id
        =None):
        # handler function to add a dataflow based on match

        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
```

Listing A.1 (Cont.): Creating a RYU controller app

```
        inst = [parser.OFPInstructionActions(ofproto.
            OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=
                buffer_id,
                                    priority=priority, match=match,
                                    instructions=inst, hard_timeout=11)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=
                priority,
                                    match=match, instructions=inst,
                                        hard_timeout=11)
        datapath.send_msg(mod)
        # self.logger.info("Flow Added")
```

To handle multicast flows, we define the following *get_buckets* function, which creates the group flow mod with action buckets corresponding to the number of devices identified as controllers.

Listing A.2 Create buckets for multicast flows

```
    def get_buckets(self, datapath, src, in_port):

        # prepare the buckets to resolve multicast
        # Each multicast packet will only be sent to the list of
            controllers

        buckets = []
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        dpid = datapath.id
```

Listing A.2 (Cont.): Create buckets for multicast flows

```
        try:
            for newdst in self.controllers:
                # Each bucket contains the respective actions of
                # rewriting the packet headers and forwarding to some
                    port
                action = []
                temp_port = self.mac_to_port[dpid][newdst]
                action.append(parser.OFPActionSetField(eth_dst=newdst
                    ))
                if (temp_port == in_port):
                    action.append(parser.OFPActionOutput(ofproto.
                        OFPP_IN_PORT))
                else:
                    action.append(parser.OFPActionOutput(temp_port))
            buckets.append(parser.OFPBucket(actions=action))
            # self.logger.info("Created buckets for src %s",src)
            return buckets
        except:
            self.logger.info("Controller device %s port unknown",
                newdst)
            self.logger.info("Flood packet on all ports")
            return -1
```

The enforcement of policies in HestiaSwitch happens at every *packet_in* event. This function is triggered whenever a new packet is logged by the open vswitch on the wireless access point. Based on the type of packet, an appropriate policy is generated and enforced by the application and installed as a flow rule on the switch.

Listing A.3 Packet_In event handler

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    # If you hit this you might want to increase
    # the "miss_send_length" of your switch
    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes
            ",
                          ev.msg.msg_len, ev.msg.total_len)

    # Extract packet data
    msg = ev.msg
    datapath = msg.datapath
    # self.delete_all_flows(datapath)
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']


    # Multicast flag
    group_flag = 0
    self.group_id = config.group_ctr
    self.controllers = config.configuration['controllers']
    self.filter = config.configuration['multicast']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    if eth.ethertype == ether_types.ETH_TYPE_LLDP:
        # ignore lldp packet
        return
```

Listing A.3 (Cont.): Packet_In event handler

```
        dst = eth.dst
        src = eth.src

        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})

        self.logger.info("packet in src%s dst%s in_port%s", src, dst,
             in_port)

        # learn a mac address to avoid FLOOD next time.
        self.mac_to_port[dpid][src] = in_port

        if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD


        # Non Controller can only multicast to controllers
        if dst in self.filter and src not in self.controllers:
            #Multicast routing
            buckets = self.get_buckets(datapath, src, in_port)
            if(buckets != -1):
                # all the controllers are known and connected
                self.group_id = self.group_id + 1
                # update group_ctr from config
                config.group_ctr = self.group_id
                group_mod = parser.OFPGroupMod(datapath,
                        ofproto.OFPGC_ADD, ofproto.OFPGT_ALL,
                        self.group_id, buckets)
                datapath.send_msg(group_mod)
                group_flag = 1
                # Group action pointing to the group identifier
```

Listing A.3 (Cont.): Packet_In event handler

```
            actions = [parser.OFPActionGroup(group_id=self.
                group_id)]
            actions.append(parser.OFPActionOutput(ofproto.
                OFPP_LOCAL))
        else:
            # if any controllers port is not known, flood but don
                't install flow
            actions = [parser.OFPActionOutput(out_port)]
            actions.append(parser.OFPActionOutput(ofproto.
                OFPP_IN_PORT))
            actions.append(parser.OFPActionOutput(ofproto.
                OFPP_LOCAL))
            group_flag = 0
    else:
        # Unicast routing for non controllers or multicast by
            controller
        actions = [parser.OFPActionOutput(out_port)]
        if out_port == in_port or out_port == ofproto.OFPP_FLOOD:
            #Send back to WLAN port
            actions.append(parser.OFPActionOutput(ofproto.
                OFPP_IN_PORT))
            actions.append(parser.OFPActionOutput(ofproto.
                OFPP_LOCAL))

    # install a flow to avoid packet_in next time
    if (out_port != ofproto.OFPP_FLOOD) or (group_flag == 1):
        if ((src not in self.controllers) and (dst not in self.
            controllers)) and group_flag!=1:
            # Check if message is to/from controller
            # self.logger.info("Communication between Non-
                Controllers Drop Flow")
            actions = []
        # else:
```

Listing A.3 (Cont.): Packet_In event handler

```
            # self.logger.info("Valid Flow!")
        self.logger.info("port::" + str(out_port))
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst,
            eth_src=src)
        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow(datapath, 31, match, actions, msg.
                buffer_id)
            return
        else:
            self.add_flow(datapath, 31, match, actions)

    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data
    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.
        buffer_id,
                              in_port=in_port, actions=actions,
                                  data=data)
    datapath.send_msg(out)
```