Securing IoT Networks Through Specification Based IDS/IPS

A MAJOR QUALIFYING PROJECT

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science in

Computer Science

by

Matthew Hagan

Ryan LaPointe

Peter Maida

March 6, 2020

APPROVED:

Professor Craig A. Shue, Project Advisor

Abstract

As consumer home automation devices are gaining popularity, more examples exist of these devices being compromised at scale to create large botnets. In this paper, we present Soteria, a specification based IDS/IPS with the aim of reducing the spread and impact of IoT botnets by blocking malicious outbound traffic from a compromised device. We demonstrate Soteria's ability to block specification-violating network traffic from IoT devices and to alert an IoT device's manufacturer and owner when the device violates a policy. Furthermore, Soteria's ability to alert the manufacturer of compromised IoT devices gives manufacturers increased awareness of vulnerabilities being exploited in the wild. We expect that with adoption, Soteria will help limit the spread and impact of IoT botnets.

Contents

1 Introduction **7**

1.1 Research Question . . . 7

2 Background **9**

2.1 Internet of Things . . . 9

2.2 Network Filtering . . . 9

2.3 OpenWRT . . . 9

2.4 Transport Layer Protocols . . . 10

2.5 Encryption . . . 10

2.5.1 Public Key Cryptography . . . 10

2.5.2 Digital Signatures . . . 10

2.5.3 X509 . . . 10

2.5.4 TLS . . . 11

2.6 Replay Attack . . . 11

3 Related Work **12**

3.1 Device Fingerprinting . . . 12

3.2 Anomaly-based IoT IDS/IPS . . . 12

3.3 Device Spoofing Prevention . . . 13

3.4 Cloud-based IDS/IPS . . . 13

3.5 Commercial IDS/IPS . . . 14

3.6 Signature-based IDS/IPS . . . 15

3.7 Cooperative Keying . . . 15

3.8 Central Firewall Policy Manager . . . 15

4 Design **17**

4.1 Design Considerations . . . 17

4.1.1 Reliability . . . 17

4.1.2 Efficiency . . . 17

4.1.3 Scalability . . . 17

4.2 Infrastructure . . . 17

4.2.1 Client . . . 18

4.2.2 Middlebox . . . 18

4.2.3 Wireless Access Point . . . 18

4.3 Advertisement Schema . . . 18

4.3.1 Policies . . . 19

4.3.2 Rules . . . 19

4.3.2.1 Selectors . . . 20

4.3.2.2 Notifications . . . 21

4.3.3 Tags 21
4.3.4 Certificate 21
4.4 Firewall 21
4.5 Cooperative Keying 22
4.6 Policy Management 22

5 Implementation 23
5.1 Infrastructure 23
5.1.1 Server 24
5.1.2 OpenWRT 25
5.2 Client 25
5.2.1 Normal Traffic 26
5.2.2 Advertisement Sending 26
5.2.3 Key Sharing 26
5.3 Advertisement Model 27
5.3.1 Signature Verification 28
5.3.1.1 MITM Attack 28
5.3.2 Parsing Advertisement into Policy 28
5.4 Database Persistence 29
5.4.1 Diesel ORM 29
5.4.2 Database Layout/Schema 29
5.5 Firewall 30
5.5.1 NFQueue 30
5.5.2 Flows 30
5.5.3 Match Status 31
5.5.4 Default Status 31
5.5.5 Tshark Cooperative Keying 32
5.5.6 Centralized Policy Manager 32
5.6 Notification Methods 32
5.6.1 REST-API 32
5.6.2 E-mail 32

6 Results 33
6.1 System Working Correctly 33
6.2 Throughput Performance 33
6.2.1 Baseline Tests 35
6.2.2 Firewall Tests 36
6.3 Memory/CPU Performance 38

7 Discussion **41**

7.1 Engineering Lessons Learned . . . 41

7.1.1 Shared Memory . . . 41

7.1.2 Libraries . . . 41

7.1.2.1 OpenSSL . . . 41

7.1.3 Cross-Compilation . . . 42

7.1.3.1 Rustup . . . 43

7.2 Future Development Considerations . . . 44

8 Conclusion **45**

A Policy XSD Schema **50**

B Test Cases **52**

B.1 System Working Correctly . . . 52

B.2 Bad Actors Introduced . . . 56

B.3 Throughput Performance . . . 57

B.4 Memory/CPU Performance . . . 58

List of Figures

1 Feature Sets of Existing Consumer Solutions 15
2 XML Policy Overview 20
3 Residential Network Layout 23
4 Testing Infrastructure 24
5 Policy UML 30
6 Database Schema 31

1 Introduction

Consumer home automation devices are rapidly gaining popularity. These devices include Internet-connected light bulbs, doorbells, power outlets, refrigerators, washing machines, televisions, speakers, cameras, and more. The firmware of such low-power, embedded systems is often at a higher risk of vulnerabilities. There are numerous examples of these devices, which we hereafter refer to as Internet of Things (IoT)[1] devices, being compromised at scale to create very large botnets. The Mirai botnet, being a prominent example, composed of consumer routers, cameras, and other IoT devices infected using default passwords, successfully performed a Distributed Denial of Service (DDoS)[2] attack on a major authoritative DNS provider knocking Twitter, GitHub, and numerous other sites offline in 2016 [1]. Multimedia centers, televisions, and refrigerators have been observed sending spam emails as part of a botnet [2]. Critical security vulnerabilities that could enable botnets have been found in light bulbs, power outlets, and IoT hubs [3, 4, 5]. After the Mirai source code was published, numerous Mirai variants have been detected.

The prevalence of insecure IoT devices is unlikely to abate because many manufacturers lack incentive to secure their devices [6]. Users usually do not consider security and the availability of firmware security updates before purchasing an IoT device, so providing security updates does not generate any revenue for the manufacturer and failing to properly secure their devices does not result in any revenue lost. Only the few most prominent manufacturers of IoT devices need to worry about brand damage from news headlines when their devices are found vulnerable [7].

Users cannot access the operating system of IoT devices by design. As such, users cannot patch security vulnerabilities in their IoT devices without manufacturer support. When manufacturers fail to properly secure their devices and to provide security patches, the devices must be protected at the network level.

1.1 Research Question

In this paper, we ask the question, *To what extent can a manufacturer aid a plug-and-play[3] security model to reduce the spread of botnets at the network level?* We answer this question by proposing an intrusion detection and prevention method that enforces a network security policy specified by the manufacturer of each IoT device.

The contributions of this work are:

1. We present Soteria, a specification based IDS/IPS with the aim of reducing the spread and impact of IoT botnets by blocking malicious outbound traffic from a compromised device.

[1] The term Internet of Things is overloaded, sometimes referring to industrial sensor networks and sometimes consumer “smart home” devices. We use it in the latter sense.

[2] A Denial of Service attack occurs when a malicious actor is using all of the available resources on a machine, denying access to legitimate users. A Distributed Denial of Service attack is similar, but its requests come from many different computers, or in this case IoT devices.

[3] Plug-and-Play refers to a model of computing where the device is automatically configured once added to the network.

2. We demonstrate Soteria's ability to alert the owner of compromised IoT devices so that they can take remedial action.

3. We demonstrate Soteria's ability to alert the manufacturer of compromised IoT devices so that the manufacturer can have increased awareness of vulnerabilities being exploited in the wild.

4. We expect that with adoption, Soteria will prevent help to prevent further IoT device exploitation's given the increased awareness on the manufacturer's behalf.

In order to have any significant impact on the spread of IoT botnets or to provide IoT device manufacturers with meaningful insight into the spread of botnets, our system must be widely deployed. Therefore the feasibility of wide deployment is a design requirement of our system.

2 Background

This research project dives deep into advanced network security concepts. Accordingly, we provide a summary of various concepts that will be referenced in later sections of the paper.

2.1 Internet of Things

Internet of Things, or IoT for short, refers to a system of interconnected computers that allow information to be transmitted between them. The IoT space encompasses industrial and residential applications. Industrial IoT is used in factories or commercial space. These applications range from oil and gas to the automotive industry. Residential IoT commonly refers to the smart home. In practice, it is any computer (typically embedded) that provides a limited scope of application by either inputting data about its surroundings or by enabling an appliance to change its surroundings. Examples of this range from a smart thermostat to smart lighting. For our project we are working exclusively in the residential space; however, our project could also apply in the industrial space.

2.2 Network Filtering

Network filtering is crucial for security. Network traffic from the Internet flows in and out of a residential network via a router on the residential network. This router is a bottleneck for all network traffic. From this point, we are able to access and filter the packets being sent in and out of the residential network. Packets can be filtered by adding different actions to the filtering table that will handle the packets. The relevant actions for this project are ACCEPT and DROP, which will either accept the network flow or drop packets from that network flow respectively. By putting our firewall system on this bottleneck, we are easily able to filter traffic either flowing into our residential network or flowing out of our residential network.

2.3 OpenWRT

Open Wireless router, or OpenWRT [8] for short, is an open-source project for embedded operating systems that are primarily used to route traffic. As an operating system, it has been configured to be able to run on constrained hardware devices, as most residential routers have both limited memory and storage. OpenWRT can be configured either via a command-line shell or a LuCI web interface. As it is an open-source environment, it has a wide range of supported packages and can be configured to fit a wide variety of networking situations, from an average router to a Wireless Access Point, or WAP for short.

OpenWRT allows users to configure their network to how they see fit. However, issues can arise from more advanced use cases, such as cross-compilation. This will be covered in more detail in the discussion section of the paper.

2.4 Transport Layer Protocols

There are two common transport layer protocols for network traffic. The Transmission Control Protocol, otherwise known as TCP, is a connection-oriented protocol where both sides need to establish a connection via a three-way handshake before the sender starts sending its data. The User Datagram Protocol, otherwise known as UDP, is a connection-less protocol that does not need to establish this connection; therefore, the sender can send its data without knowing that the receiver is capable of receiving this information. TCP is used when the information sent needs to be received correctly and fully. UDP is used when it is allowed for some of these packets to be lost, damaged, or out of order. UDP is the faster option because the three-way handshake does not need to occur and is used for applications such as voice chats, video chats, and online gaming.

2.5 Encryption

Encryption primarily addresses the security goal of confidentiality. We outline some of the encryption methods and techniques that our report addresses later on.

2.5.1 Public Key Cryptography

Public key cryptography uses a pair of keys to encrypt and decrypt data. Depending on the use case, one of these keys will be kept secret and the other will be disseminated to the intended audience. For encrypting a message, the sender would use the receiver's public key. Then, to decrypt the message in question, the private key would be used. In this situation the private key is kept private to the receiving party. As public-key cryptography is fundamental to network infrastructure in the modern day, there are a lot of standards used. In this report, we will primarily be relying on TLS, RSA, and digital signatures with X509 Certificates.

2.5.2 Digital Signatures

Digital signatures are a way for a user or entity to validate both that the sender is who they say they are and that the sent message was not altered in transit via man-in-the-middle (MitM) attacks. This form of authenticity and integrity is the other large part of Public Key Cryptography, as it relies on a private key being disseminated and the public key kept secret.

2.5.3 X509

An X509 certificate is a standard public key certificate used in a variety of Internet protocols, including TLS. An X509 certificate contains both an identity and a public key. This identity can be either a hostname or an organization name or an individual. When a certificate is signed by a trusted certificate authority, the using party can rely on the certificate's public key to establish secure connections with another party or validate digitally signed documents with the private key.

2.5.4 TLS

The Transport Layer Security (TLS) protocol encrypts data on the Internet to provide confidentiality, integrity, and authenticity. Traffic being sent over an insecure channel is encrypted with a symmetric key on both ends to provide confidentiality. The newer version of TLS that is used in this project, TLS 1.3, uses the Diffie Hellman algorithm when performing the initial handshake to determine the symmetric key to be used. TLS provides integrity by calculating a message digest with a hashing algorithm that is verified on the receiving end. Finally, TLS provides authenticity by signing the data with the receiving end's public key, to be verified with the receiving end's private key upon arrival.

2.6 Replay Attack

A replay attack, also known as playback attack, is a form of network attack in which a valid data transmission is maliciously or fraudulently repeated or delayed [9]. Attackers "replay" a previous network transmission to manipulate the system without having to construct a valid message themselves. This allows bad actors to take advantage of a system in order to identify themselves as another party. Similar to a replay attack, a downgrade attack will transmit a valid older version of the code with known vulnerabilities in order to downgrade the software version and exploit previous vulnerabilities.

3 Related Work

In this section, we examine existing intrusion detection and prevention systems for IoT devices.

3.1 Device Fingerprinting

IoT Sentinel [10] is a system designed to automatically identify the type of device connected to a network and enforce rules for constraining its communication such that vulnerable devices can cause minimal damage in a worst case scenario. It fingerprints a device by examining initial and idle device traffic and then performs mitigation strategies such as network isolation, traffic filtering, and user notification. IoT Sentinel utilizes software defined radios (SDRs) to allow for a flexible protocol at the cost of increased overhead to the system.

IoT Sense [11] continued the work of IoT Sentinel. The authors found that having protocol flexibility makes it difficult to observe all possible protocol interactions. Beyond that, classifying a device's behavioral features is not a trivial process and may require things like packet header and payload features. Finally, generating a statistical model on device behavior is difficult with multivariate data, as it requires a large quantity of data for device certainty.

While IoT Sentinel and the related research of IoT Sense are aimed at fingerprinting using network traffic, S2M [12] is an acoustic hardware fingerprinting method. The authors' approach is to use the minute differences in speaker and microphone manufacturing processes to identify a device based on its noise response. The downsides are that this solution can be cumbersome when the response is being sent and received, and it requires that the IoT device has both a speaker and microphone.

These solutions rely on various methods such as network filtering and pattern recognition to fingerprint a device. Our solution will utilize a chain of trust from a trusted certificate authority to ensure that the client maintains integrity.

3.2 Anomaly-based IoT IDS/IPS

IoT-IDM [13], an intrusion detection and mitigation system, uses a customized machine learning model to detect anomalies within the network. Users are able to define the features used in the model, but to do so requires an extensive knowledge of the hosts and possible attacks. In addition, since the sensor element of IoT-IDM is on top of the SDN controller and unable to handle the high volume of residential network traffic, it is only feasible to include select IoT devices that do not add a lot of overhead on the SDN controller.

Golomb et al. [14] proposed CIoTA, a collaborative IoT anomaly detection system via blockchain. Their main premise is to utilize a blockchain of trust to perform both distributed and collaborative anomaly detection on devices with limited resources. This blockchain allows for a self-attestation and consensus among IoT devices in the network. The primary drawback of this system is that it assumes there are a large number of IoT devices within a network, which is not always the case.

Apthorpe et al. [15] argued that an ISP or network observer can infer different traffic interactions, even in encrypted space. The team achieves this by utilizing a variety of things, from the MAC addresses, DNS queries, and traffic rate models over time. Beyond that, they argue that traffic shaping is a valid method in reducing security risks. Traffic shaping disperses traffic instances into a single load. While this does create more anomalies between devices, it introduces new problems in network traffic collision, such as the IEEE 802.11 Hidden Terminal problem.

Cho et al. [16] proposed detecting botnet-infected sensor nodes on a 6LoWPAN by monitoring packet length, number of connections, and sum of the TCP control field on the traffic of each node for anomalies. The authors achieved a low false positive rate, but made very narrow assumptions that do not apply to this project.

Anomaly-based intrusion detection or prevention systems are able to predict a firewall action for network traffic with a high accuracy. Unlike these, our system will achieve a 100% firewall accuracy by not using predictive machine learning models.

3.3 Device Spoofing Prevention

Liu et al. [17] described an intrusion prevention system for home IoT devices that consists of two parts: first, requiring each client that wishes to communicate with an IoT device to obtain temporary authorization from a server using a time-based one-time password (OATH-TOTP) [18]; and second, performing deep packet inspection on all traffic to IoT devices and blocking packets that do not conform to a pre-generated state machine model of the IoT device's normal application-layer traffic. The TOTP-based authentication scheme requires active involvement from the consumer. Similar to our design, the specification-based protocol validation requires the firewall to have a preexisting model of every possible protocol, which necessitates service provider support as new IoT devices and protocols are created.

To prevent ARP spoofing in the data link layer, Data et al. [19] proposed an automatically managed static ARP cache table for MAC and IP addresses. Their solution is compatible with the standard ARP protocol, is easy to implement on hosts, and has been tested to protect hosts against all types of ARP spoofing attacks. A minor shortcoming with this solution is that hosts are unable to communicate with other hosts to alert them of an ARP spoofing attack within the network.

Since advertisements will not be changed regularly, we are unable to add the security of temporary passwords in our system. Therefore, our solution plans to use certificate authorities to prevent device spoofing.

3.4 Cloud-based IDS/IPS

Feamster [20] proposed that the security of home and small business networks should be remotely managed by a service provider because consumers and small businesses cannot afford to manage their networks properly. He proposed to use OpenFlow software-defined networking for performing remote management.

Taylor et al. [21] built upon the idea of outsourcing security management of consumer LANs, proposing a system to enable the flexible proxying of traffic from individual applications through different security service providers.

Taylor et al. [22] also presented a middlebox-based system, TLSDeputy, that ensures residential networks only connect to properly verified TLS servers by monitoring the TLS handshake process and performing independent verification of TLS certificates and revocation checking using certificate revocation lists.

Gajewski et al. [23] proposed that Internet Service Providers take charge of securing home IoT devices through a software module on the home Internet gateways provisioned to consumers, backed by processing on the ISP's servers.

Sivaraman et al. [24] proposed outsourcing network security operations for home LANs to a third party "Security Management Provider" (SMP) which would be located between the ISP and home network, and add new competition to the security industry.

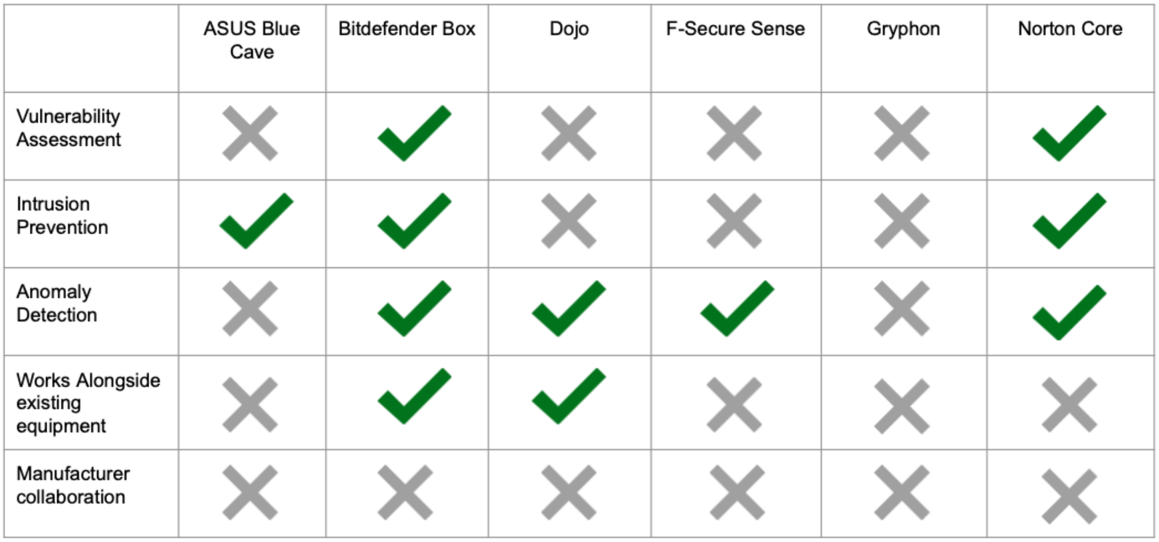
In "Making Middleboxes Someone Else's problem: Networking Processing as a Cloud Service," Sherry et al. [25] examine the existing issues with the middlebox infrastructure to ultimately propose APLOMB. Through their findings, they conclude that outsourcing the processing can help to lower costs, complexity, and failures. While these results are promising, it is designated for a corporate setting, not a residential one.

These solutions either require that the consumer pay for a third party network security service, cloud hosting, or require Internet service providers to provide network security management as an included benefit of their home Internet service. Although there is enough consumer demand for third party network security services to sustain several commercial offerings in that area [26, 27, 28, 29, 30, 31], some consumers may be unwilling to pay for a network security service. It is unlikely that every Internet Service Provider will provide meaningful network security management at no extra charge to their customers.

With this additional overhead in mind, we plan to make it easier for both the consumer and manufacturer by not forcing the consumer to pay for any third-party services to use our solution, as it can be run natively on the router and manufacturers don't have to negotiate service deals with third-party providers to use the service.

3.5 Commercial IDS/IPS

We note the various features and requirements of other commercial products in the field with the table in Figure 1. The solutions referenced in the Figure are Asus Blue Cave [26], Bitdefender Box [27], Dojo [28], F-Secure [29], Gryphon [30], and Norton Core [31]. Of all of these existing consumer solutions, none assume manufacturer collaboration. Manufacturer collaboration allows for the middlebox involved to have a more informed sense of intended device operations. Beyond that, it allows for manufacturers to obtain "security compliance" and pass on the responsibility of secure network traffic of their devices by complying with the given security solution. Given the benefit that manufacturer collaboration can add to a residential network, it is something that we plan on incorporating into our project.



	ASUS Blue Cave	Bitdefender Box	Dojo	F-Secure Sense	Gryphon	Norton Core
Vulnerability Assessment	✗	✓	✗	✗	✗	✓
Intrusion Prevention	✓	✓	✗	✗	✗	✓
Anomaly Detection	✗	✓	✓	✓	✗	✓
Works Alongside existing equipment	✗	✓	✓	✗	✗	✗
Manufacturer collaboration	✗	✗	✗	✗	✗	✗

Figure 1: Feature Sets of Existing Consumer Solutions

3.6 Signature-based IDS/IPS

Von Sperling et al. [32] created an IoT router that verified the DNS traffic originating from IoT devices to detect devices consulting with unauthorized DNS servers. The monitored traffic is sent to an embedded software analyzer to verify the DNS protocol flows in search for indicators of an unauthorized DNS server.

Al-Bahri et al. [33] proposed a lightweight signature-based technique for anomaly detection. Through the use of game theory and the Nash equilibrium, anomaly detection is only activated when a new attack's signature is predicted to occur. While the concept is novel, sacrificing detection accuracy for CPU time may allow any unaccounted breach in a network to result in the entire network being compromised.

3.7 Cooperative Keying

Bierma et al. [34] presented Locally Operated Cooperative Key Sharing (LOCKS), a system that enables local clients to share their TLS session keys with a security monitoring system to facilitate deep packet inspection without breaking the end-to-end authentication component of TLS. The client side sends its TLS session key to a trusted agent, which then shares the key with the IDS. The decryption in the LOCKS system adds overhead within manageable limits. We plan on utilizing the LOCKS framework as the added benefits of deep packet inspection will allow for the middlebox to inspect the network traffic while maintaining packet integrity.

3.8 Central Firewall Policy Manager

Hachana et al. [35] augmented the traditional approach of mining policies from several firewalls into one policy by adding additional processing for network access control policy mining. Their approach follows a bottom-up framework that runs each policy separate, merging the policies after they have been mined. They mention that using high abstract models for controlling policies is more efficient and safer than manipulating the firewall configuration at a low level. Similarly, our project runs multiple policies separately and takes the most restrictive action, rather than breaking apart policies.

4 Design

In this section, we describe the design process of Soteria, our specification-based IDS/IPS. We begin with the design considerations that were made and the various constraints that we had to account for, such as reliability, efficiency, and scalability. Following that, we explain how we set up our infrastructure, schema, firewall, cooperative keying and policy management.

4.1 Design Considerations

In our Introduction section, we explained that the goal of this project was to reduce the spread of botnets at the network level with the manufacturer's help. With this in mind, it is important that the system we design meets the following goals of reliability, efficiency, and scalability.

4.1.1 Reliability

Reliability is an important aspect of this project. Our system needs to be stable enough that it can be run without crashing. Beyond that, as this project security-oriented, our system needs to add to the security of the network and not introduce any weak points within the network. As such, we investigated design languages such as Go and Rust but ultimately decided on building our system in Rust because it guarantees memory safety and segfault free operation.

4.1.2 Efficiency

Efficiency was also important for this project. As our system is functioning on a routing level, it should not introduce latency or throughput problems on existing low resource systems found in most residential routers. Given this constraint, we as a team made sure to implement coding techniques such as threading and take advantage of Rust's run time performance guarantees.

4.1.3 Scalability

Finally, scalability was an important goal for our system. The reason that this is important for our project is that this system was designed to be implemented by multiple users, manufacturers and third-party security providers, therefore making it important that the system we inevitably designed was able to scale and meet their needs. To this end, we looked for ways to make our system easy to scale to other developers, such as utilizing object-relational mappings for the database or continuous integration/continuous deployment testing for the project as a whole to see what extent can a manufacturer aid a plug-and-play model.

4.2 Infrastructure

With these considerations in mind, we moved forward with the design of our infrastructure. To make it easy for continuous integration/continuous deployment testing, we have decided to deploy our test network on virtual machines. The use of virtual machines would also give us a large amount of control over the “hardware” restrictions of both the client and the middlebox, allowing us to scale our test infrastructure as we saw fit.

4.2.1 Client

Within our test network, there are various modes of operation we are trying to simulate. The code that will be written for the client represents all of the various IoT interactions within a network. Because it is a simulation, it needed to function somewhat like an actual IoT device within a network.

The client’s primary goal is to simulate normal traffic and send advertisements. Because this device is not making decisions for the network (unlike the middlebox), language efficiency is less of an issue. As such, the team decided to write the client’s codebase in Python [36]. The reason for this is that Python’s libraries are extensive, allowing for rapid deployment that is both stable and efficient. Beyond that, Python code testing can be done very thoroughly. Utilizing the Tox [37] tool with the PyLint [38] and Flake8 [39] plugins, the team can ensure code is standardized.

4.2.2 Middlebox

The middlebox is where all the policies for each device on the network are stored and applied. It is intended to be run on the router in order to have access to different VLANs on the network.

When choosing the programming language for the middlebox, our main concern was performance. C would allow for the near ideal performance; however, we had additional security concerns that by programming in C, our code would contain inevitable bugs and vulnerabilities.

Rust claims an easy way to write secure code. Rust is completely memory safe, lets programmers know how their memory is managed every time the program runs and has a strict compiler to assure that various run time errors do not occur [40]. Therefore, we went with the newer language Rust for both performance and security benefits.

4.2.3 Wireless Access Point

For this system, we needed a way for existing and simulated IoT devices to be able to connect to our system. As such, we set up an access point to allow devices to connect to our server so that we could perform a real-world test of Soteria to see proof that it performed the tasks that we set out for it to do.

To set up our Access point, we needed an operating system that was easy to control and offered a wide range of customization. This is where OpenWRT came in, as it allows for the administrator to configure a stock router and a wide variety of ways, some of which the manufacturer does not support out of the box.

4.3 Advertisement Schema

The advertisement messages we designed follow the HTTP text-based format, chosen for easy user use as well as easier development. Advertisements have a header section for meta data and a body section for the payload of the firewall policy. The meta data includes information that this packet is an advertisement, the version of Soteria it is using, and the signature for the body of the packet. It is important to distinguish the advertisement packets from other packets being sent through the system, so they can be parsed differently by the firewall. The version of Soteria will be used for updates to this software when it becomes commercially viable. The signature is important in verifying that the advertisement hasn't been tampered with. The payload of the advertisement is an XML file with the schema outlined in Appendix A, further detailed by the sections below. Advertisements are in XML format because of the simple user usage as well as the simple parsing to policy for the system. These advertisements are designed to be multicasted over the network frequently with UDP multicast so that joining devices on the network will be updated to other devices on the network. By frequently multicasting these advertisements, no additional tracking needs to occur for devices currently on the network and their corresponding policies.

4.3.1 Policies

Policies are in XML format with a schema outlined in Appendix A. A diagram of what an example policy consists of can be seen in Figure 2. A policy is identified with a universally unique identifier (UUID) and revision number from the party that created the policy. When a policy gets updated, the same UUID must be used with an increased revision number. Different policies are distinguished via their UUID's. We have chosen a UUID system because it allows for multiple parties to have a unique ID without needing an additional system to distribute ID numbers to company policies. We have included a revision number with the policy to negate replay attacks. A policy will only be applied if the new revision number is greater than the current revision number, therefore any repeated, replayed, or downgraded advertisements will not be applied by the middlebox. This partially satisfies one of our design goals of security. Nested inside a policy are any number of tags, any number of rules, and a single certificate. Each of these nested blocks are explained further below.

4.3.2 Rules

The main purpose of a policy are rules. Rules dictate an action if a certain packet should be accepted or dropped. This gets decided with various conditions in the rule. Multiple conditions can be included in a rule. The attributes inside of conditions all need to be true in order for the condition to be true. In other words, the attributes, which we call selectors, inside of a condition are a logic “AND” together. However, each condition within a rule are a logic “OR” together so that only one condition needs to be true in order for the entire rule to be matched to the packet it is tested against. In addition to conditions, multiple notifications can be included in a rule so that the user or manufacturer can be notified if an error has occurred or a certain rule has been met. Rules in a policy are prioritized top down so that the first rule that matches is applied.

Rules contain an additional condition nesting to abstract out the similarities inside a rule. Rather than a manufacturer repeating the same action and notifications for a group of selectors, the group of selectors is simply added to a condition. Conditions have a logical “OR” similar to how they would otherwise have been applied if only one condition and group of selectors were added to a rule.

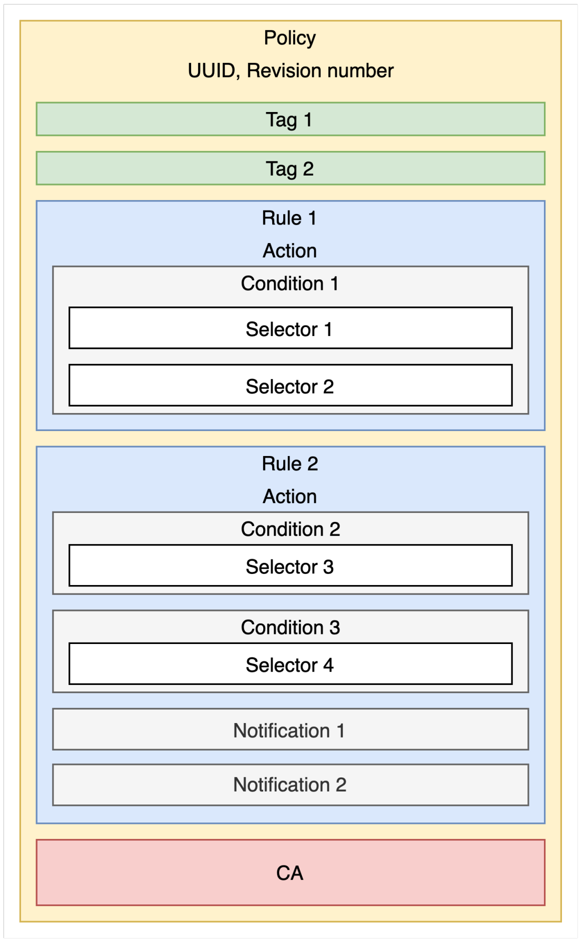


Figure 2: XML Policy Overview

Otherwise the selectors inside the condition have a logical “AND” to allow more specific policy rules. For example, a condition that is able to match on a specific IP address at a specific port.

4.3.2.1 Selectors

Selectors are the attributes in a condition that get logically “ANDED” together. The selectors we have chosen to include are the IPv4 remote address, IPv6 remote address, TCP remote port, and UDP remote port. The remote device is the device the IoT device is communicating with, whereas the source device would be the IoT device itself. These are the selectors that we were able to implement, but other selectors could include autonomous systems, HTTP host headers, TLS CAs, and options for incoming traffic to the source IoT device. We decided on implementing the outlined IP address and port selectors due to their simplicity and ability to show the proof of concept of our firewall policy system.

4.3.2.2 Notifications

Notifications are included in a rule at the same depth of conditions. The notifications we have implemented include email notifications and REST API calls. If one of the conditions in the rule is meet, all of the notifications in the rule will be sent to their various destinations. This allows for multiple parties to be notified of the traffic being received in devices, including bug instances where a rule is applied when it shouldn't be reached. We included these two notification methods for easy implementation with libraries, but other methods could include an app that provides user alerts to the device with a security vulnerability.

4.3.3 Tags

Policies can also include various tags. A tag is a way to group various IoT devices together. Tags serve two purposes: allowing policies to refer to groups of devices and allowing appropriate centrally-managed policies to be applied. For example, a light bulb may advertise a policy that allows it to talk to other light bulbs on the LAN, identifying those light bulbs by tag. Additionally, with the presence of tags manufacturers can included a specific tag in their policy to include all of the rules of that tag, rather than coding those rules themselves. An example of this could be a tag from Google that passes all valid Google traffic on specific IP addresses and ports.

4.3.4 Certificate

Policies always contain a single certificate. This certificate establishes a root of trust for the signatures of future advertisements from the same client. A policy's certificate will be validated by a certificate authority to prove its authenticity. With the inclusion of this certificate, we satisfy another security goal design consideration.

4.4 Firewall

A packet arriving at the middlebox to/from a LAN host that has never sent an advertisement or been manually added to the middlebox's list of clients will be allowed by our filtering. A packet to/from a host that is a registered client will be dropped unless there is a policy rule allowing it. This allows devices that do not implement our protocol to coexist with devices implementing our protocol on a LAN and ensures that no malicious traffic is allowed to or from registered clients. In the event of a conflict between policies, where one policy has a rule that dictates to drop a packet and another policy has a rule that would pass the packet, the packet is dropped. By applying the stricter rule to our firewall, a party that fails to block insecure traffic is overridden by the party that successfully blocks insecure traffic. In addition, a malicious actor that is able to add a policy to the firewall can only deny service to/from the user's device. The actor is unable to infect the user's device and add it to their botnet.

4.5 Cooperative Keying

The middlebox includes functionality to receive TLS key information from clients to enable it to decrypt those clients' HTTPS connections. The middlebox inspects the HTTP host headers of HTTPS connections and uses that information when applying selectors in a policy. The ability to decrypt HTTPS connections is important because otherwise HTTPS provides a mechanism for malicious traffic to be shielded from inspection.

4.6 Policy Management

Some users may want to use security service providers to protect their networks. Our system has a facility for a third party to provide additional policies. When the middlebox receives an advertisement from a client, it uses the tags contained in the advertisement to fetch managed policies from a central policy manager. The firewall applies both the policies sent by the client in its advertisement and any centrally-managed policies that apply to the tags sent by the client.

5 Implementation

In this section, we will describe the implementation of Soteria and the various features achieved. We begin with the infrastructure created looking both at the server and the wireless access point. We then move onto the client, advertisement model, Database Persistence, the firewall, and notification methods.

5.1 Infrastructure

Our approach requires each IoT device to periodically multicast advertisement packets that contain the manufacturer's firewall policy specification. We decided to multicast these packets rather than broadcast as it allows for a more fine-grained control of who the receiving party should be. These advertisements are received and cached by firewalls, routers, and IDS sensors, which then enforce the specified policy for each device. We hereafter refer to the devices receiving and caching the policies as firewalls. Figure 3 shows the basic residential network layout of our system. There are various parts to this diagram. First, the client pool is what we will use to refer to all of the IoT devices on the network. Each device within the client pool will multicast an advertisement periodically with information about the device, such as the firmware version or the various DNS domains that it talks to. This advertisement will then be received, cached, and applied on the middlebox. From that point on, given that the advertisement is valid, any traffic that device made will follow the policy created from the advertisement.

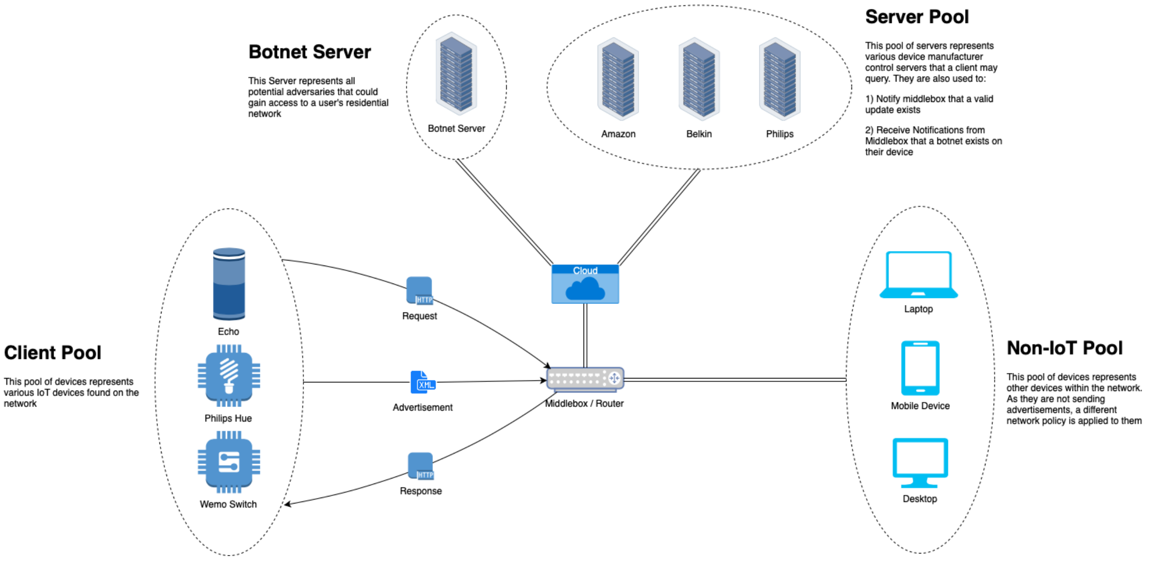
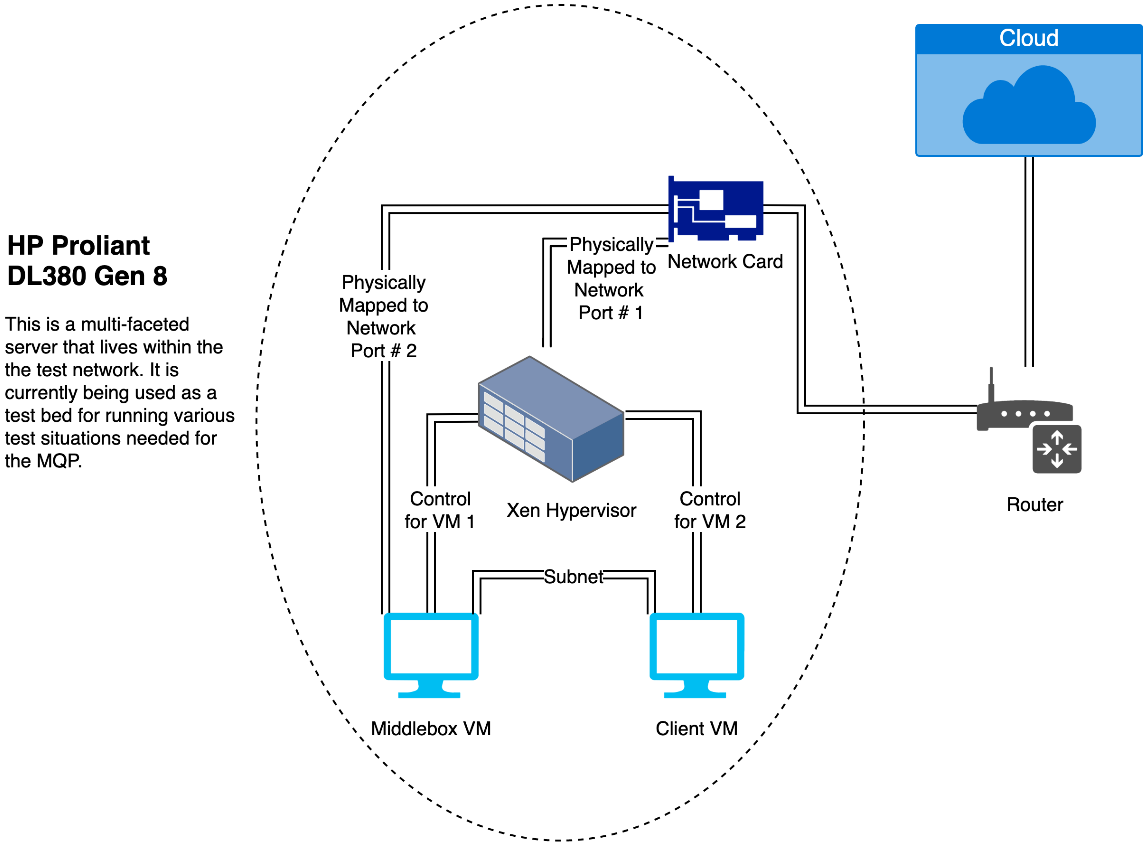

Botnet Server
This Server represents all potential adversaries that could gain access to a user's residential network
Botnet Server
Amazon
Belkin
Philips
Server Pool
This pool of servers represents various device manufacturer control servers that a client may query. They are also used to:
1) Notify middlebox that a valid update exists
2) Receive Notifications from Middlebox that a botnet exists on their device
Cloud
Client Pool
This pool of devices represents various IoT devices found on the network
Echo
Philips Hue
Wemo Switch
HTTP
Request
XML
Advertisement
HTTP
Response
Middlebox / Router
Laptop
Mobile Device
Desktop
Non-IoT Pool
This pool of devices represents other devices within the network. As they are not sending advertisements, a different network policy is applied to them

Figure 3: Residential Network Layout

With the policy applied, the client pool can then send and receive traffic as normal as long as that traffic is properly outlined in the policy. Devices that never sent an advertisement will not have any firewall rules applied and traffic will continue to be forwarded as normal. This includes the non-IoT pool from Figure 3 that does not send an advertisements to the firewall. Computers and mobile devices that connect to the router do not have to conform to this policy and may continue regular use.

The router is also connected to the Internet outside the residential network. This is where the company servers will be as well as any adversary botnet server. Any traffic that enters or leaves the router, including that to the outside Internet, are verified to be intended after being approved by the firewall policy. By applying this policy on the router, we can confirm that our goal of blocking malicious outbound traffic from a compromised device is complete.

5.1.1 Server



HP Proliant DL380 Gen 8
This is a multi-faceted server that lives within the the test network. It is currently being used as a test bed for running various test situations needed for the MQP.
Cloud
Network Card
Physically Mapped to Network Port # 1
Physically Mapped to Network Port # 2
Xen Hypervisor
Control for VM 1
Control for VM 2
Subnet
Middlebox VM
Client VM
Router

Figure 4: Testing Infrastructure

Within an HP Proliant DL380 Gen 8, our test network is composed of a series of VMs that would simulate what would occur on a normal residential network. Within that network, the Xen Hypervisor is mapped to physical Ethernet port #1. We chose to have the infrastructure work this way to allow for convenient system administration. Next, the middlebox VM is mapped to physical port #2. Within that physical port is a virtual subnet that the client lives on. This virtual subnet allows us to deploy a multitude of test devices without being limited to the physical ports of our device.

5.1.2 OpenWRT

As discussed in the background and design sections, OpenWRT is an opensource wireless routing OS designed to allow the network owner to have more control of the hardware that they buy. To get OpenWRT on the router, we followed the floodlight tutorial[41] outlined by Yu Lui. To flash the router with OpenWRT, the user needs to access the stock web portal of the router and go to the firmware update section. on the TP-link Archer C7 routers provided by the Network Security Lab, this was at http://192.168.1.1. The OpenWRT can be built from source, however, we found that there was already a binary compiled for our router once found on OpenWRT's table of hardware[42]. Once downloaded, it did not load properly onto the router. We later found that the binary needs both the OEM bin file name and the word 'factory' in it. for us, this ended up being "openwrt-ar71xx-generic-archer-c7-v2-squashfs-factory-us.bin". once this was done, we were able to load the router with the OpenWRT firmware and subsequently access the openWRT web portal to customize the router as we saw fit.

Based on our design considerations, we ended up setting up the OpenWRT flashed router as a Wireless Access Point (WAP). This WAP would allow for existing IoT devices to connect to our lab network without compromising the existing residential network. To accomplish this goal, we followed OpenWRT's Dumb AP guide[43] but did not disable the DCHP settings, as we needed our AP an IP address to the IoT clients within the test network subnet.

5.2 Client

The first thing needed to get the client up and running is Python 3. Real Python has outlined a simple guide to install it on a variety of operating systems at https://realpython.com/installing-python/. Using Python 3 will allow the team to take advantage of the vast amount of existing libraries to ensure rapid deployment that is both stable and efficient. Beyond that, Python code testing can be done very thoroughly utilizing Tox with the PyLint and Flake8 plugins.

Once Python 3 is installed, A virtual environment should to be created. Virtual environments allow users to create their own isolated site directories, such that only the required packages and nothing more need to exist in the environment. A virtual enviorment can be created by typing the following: `python3 -m venv /path/to/new/virtual/environment`. Once the environment is created, the next step is to source the environment. This is a way of loading into the environment with the required packages such that when Python scripts are run they will work as desired. This is done by typing `source environment_name/bin/activate`. Unsourcing the environment can be done by typing `deactivate`.

While in the environment, there are a few packages that should be installed if the environment is to function properly. To make it simple for the user, a requirements.txt file has been made to direct pip, or the preferred installer program, which packages to install. To install the packages, one types `pip install -r requirement.txt`. pip now comes standard with Python 3.4 and after. Once this is done, running the project can be done by typing python3 -m iot_client_sim. This will run the main.py file which will then trigger the normal_traffic.py and send_advertisement.py on regular intervals.

5.2.1 Normal Traffic

The normal_traffic.py file is responsible for generating normal traffic on the network. It will first send an HTTP get request to Google.com two times. Each time, it will print the response. After that, it will send an HTTP get request to Python.org three times. Again it will print the responses. These domains were chosen as they should maintain near perfect up time due to their usage. To create the requests, we used the HTTP and JSON packages in the python libraries to create a JSON request and then send the request to the server. Beyond that, we also used the time library to space out when our client would send messages from these different domains, as to look more realistic in the eyes of the middlebox.

5.2.2 Advertisement Sending

The second file is send_advertisement.py. This file is responsible for sending the advertisement.xml over the network using a UDP multicast. To achieve this with the proper certificates and signatures, the advertisement has been packed into an HTTPU query. This file used the ast, socket, struct, base64 and a variety of Crypto python libraries to opperate. To test that the advertisement is able to be updated, there are two versions of the sending code with different advertisements. The program flow is such that these advertisements will be first read from there .xml file. once these advertisements are read from their file, the program generates a SHA256 hash around the advertisement's body and using the server's private key generates a signature from the key and the hash. once this signature is made, it is encoded to base64 so that it can be properly sent through socket programming. at this point, the message to be sent is concatenated with the signature and then sent to the multicast group. the multicast group chosen was 192.169.90.1 with a port of 8888 such that the middlebox was able to recieve and parse the advertisement.

5.2.3 Key Sharing

The final file was key_sharing.py. This file, utilizing the sslkeylog Python library, was responsible for capturing TLS key information when the client code makes HTTPS connections. The TLS key information was transmitted to the middlebox to enable decryption of the TLS sessions. A function was registered with the library to receive the TLS key information and send it via UDP packets to the middlebox. The key information is in the SSLKEYLOGFILE format that can be produced by the Chromium and Firefox web browsers and consumed by tshark. Inspection of the client's TLS connections allows the middlebox to better protect the client by preventing the use of HTTPS to conceal malicious traffic.

5.3 Advertisement Model

The advertisement model is designed as an HTTP text-based packet. An example header of the advertisement looks like the following:

```
ADVERTISEMENT SOTERIA HTTP/1.1
signature: -----BEGIN SIGNATURE-----
          MIIDQTCCAimgAwIBAgITBmyfz5m/jAo54vB4ikPmljZbyjANBgkqhkiG9w0BAQsF
          ADA5MQswCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBBbWF6
          b24gUm9vdCBDQSAxMB4XDTE1MDUyNjAwMDAwMFoXDTM4MDExNzAwMDAwMFowOTEL
          MAkGA1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEZMBcGA1UEAxMQQW1hem9uIFJv
          b3QgQ0EgMTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALJ4gHHKeNXj
          ca9HgFB0fW7Y14h29Jlo91ghYPl0hAEvrAIthtOgQ3pOsqTQNroBvo3bSMgHFzZM
          9O6II8c+6zf1tRn4SWiw3te5djgdYZ6k/oI2peVKVuRF4fn9tBb6dNqcmzU5L/qw
          IFAGbHrQgLKm+a/sRxmPUDgH3KKHOVj4utWp+UhnMJbulHheb4mjUcAwhmahRWa6
          VOujw5H5SNz/0egwLX0tdHA114gk957EWW67c4cX8jJGKLhD+rcdqsq08p8kDi1L
          93FcXmn/6pUCyziKrlA4b9v7LWIbxcceVOF34GfID5yHI9Y/QCB/IIDEgEw+OyQm
          jgSubJrIqg0CAwEAAaNCMEAwDwYDVR0TAQH/BAUwAwEB/zAOBgNVHQ8BAf8EBAMC
          AYYwHQYDVR0OBBYEFIQYzIU07LwMlJQuCFmcx7IQTgoIMA0GCSqGSIb3DQEBCwUA
          A4IBAQCY8jdaQZChGsV2USggNiMOruYou6r4lK5IpDB/G/wkjUu0yKGX9rbxenDI
          U5PMCCjjmCXPI6T53iHTfIUJrU6adTrCC2qJeHZERxhlbI1Bjjt/msv0tadQ1wUs
          N+gDS63pYaACbvXy8MWy7Vu33PqUXHeeE6V/Uq2V8viTO96LXFvKWlJbYK8U90vv
          o/ufQJVtMVT8QtPHRh8jrdkPSHCa2XV4cdFyQzR1bldZwgJcJmApzyMZFo6IQ6XU
          5MsI+yMRQ+hDKXJioaldXgjUkK642M4UwtBV8ob2xJNDd2ZhwLnoQdeXeGADbkpy
          rqXRfboQnoZsG4q5WTP468SQvvG5
          -----END SIGNATURE-----
```

The first line of the header includes the word “ADVERTISE” to show that the current packet is an advertisement. The second word in the first line is the name of our system, Soteria, that has been included to distinguish our implementation from any other implementation. The third word in that line is the HTTP protocol version that the advertisement packet is following. The second line of the header is the signature used to verify the validity of the advertisement.

The body of the advertisement packet is the XML document that will be parsed into the policy. The XML schema described in Appendix A is the model that the policy will follow. This model includes a single policy that consists of a sequence of rules, a sequence of tags, and a CA. A rule consists of a single action, a sequence of conditions, and a sequence of notifications. A condition consists of a sequence of selectors, all of which need to be matched in order for the condition as a whole to be matched. Once a condition is matched, the sequence of notifications as well as the match action pertaining to the given rule will be applied.

5.3.1 Signature Verification

When receiving the advertisement packet, we first break apart the header and body. We parse out the signature from the header and use it to verify the body of the advertisement. The body was encoded with SHA256 before it was signed, so we need to encode the body with SHA256 again in order to verify it with the signature to see if the advertisement has not been modified. As with typical verification functions, our function takes the message body to be verified, the signature, and the public key to prove that the message has not been tampered with.

5.3.1.1 MITM Attack

A man-in-the-middle attack is where a bad actor is placed in between two systems. In our case, they would be placed between the IoT device sending an advertisement, and the middlebox on the residential router. Without signature verification, our middlebox would be unaware when the bad actor modified an advertisement. The bad actor would be able to view all of the traffic in between the two systems, extracting any sensitive information. They would also be able to modify the traffic being sent to either system, adding malicious payloads to valid traffic. Man-in-the-middle attacks can be prevented by encrypting and signing traffic. Encrypted traffic prevents packet sniffing, and signature verification prevents packet modifications.

5.3.2 Parsing Advertisement into Policy

In order to parse the HTTP text string into a valid XML policy, we used the roxmltree Rust library. We chose this Rust library over others for its efficiency. Roxmltree is faster than its counterparts because it is read-only and uses xmlparser which is many times faster than xml-rs [44].

To start, we create a Policy object that will store the content of the XML. Then, roxmltree converts the given string to a Document object which contains nested nodes for the tags used at different levels. By looping through the children of the root element of the document, we can traverse the various tags, rules, and the ca. Each one of these loops will either add the tag or ca node content to the policy object. If the child was a rule node, a Rule object gets created with the action in the rule's tag. We then loop through all of the rule node's children as well to get each notification and condition. Notifications for email and rest-api are supported and are easily added to the Rule object with an add_notification() function. All of the supported selectors in a condition are added with a similar add_selector() function. The currently supported selectors are ipv4-remote-addr, ipv6-remote-addr, tcp-remote-port, and udp-remote-port. When traversing the Document object, any unknown nodes will cause the program to panic with an error printing out the node that caused the issue. The documentation for roxmltree says that it does not support comments, but upon inspection of the library's code library, there is a function to check if a node being traversed is a comment. Using this function, we ignore any comment nodes we run into at any level of the XML.

5.4 Database Persistence

Database persistence is an important factor in this system. If the middlebox lost power or stopped running, we would want it to apply the policies as it did before it stopped running. If during a potential downtime, any of the IoT devices in the network were compromised and began broadcasting nefarious advertisements, a non-persistent system would be unaware of the invalid advertisement and apply it as if it was intended. The outcome of this potential vulnerability would be a successful subversion of our security system. Therefore, we decided that implementing a database within our system was a critical feature that would address policy integrity issues when the system is rebooted.

5.4.1 Diesel ORM

For our database, we pursued various options. Initially, we implemented a database with SQLite [45] and created database access objects using the library Rusqlite [46]. However, we found that to make it easier to scale to any potential need of our system, we needed to use an Object Relational Mapping (ORM). An ORM allows developers to query and manipulate data within a database without having to write tedious SQL queries, and instead, generate statements using code that is native to the primary development language. This allows for not only efficient queries, but also safe ones. The ORM library we decided to use was Diesel [47]. Diesel is a Rust native ORM. It is built for performance, is productive and extensible, and is good at preventing run-time errors. With it, the user can build simple or complex queries with less boilerplate.

The only concern with Diesel is the support for it. The developer is actively supporting it, but since Rust is a newer language with a developing community, there is not extensive documentation and support online. As such, we as a research team have spent more time then we were expecting to get it to work; however, were excited by the results it brought.

5.4.2 Database Layout/Schema

For the creation of the database, we begun by planning out our design via a Unified Modeling Language (UML) diagram as seen in Figure 5. At the center of the UML diagram is a Policy, the object that caused us to create the database. A policy object is made up of many rules, and these rules contain many notifications. A policy is also designed for many different clients. In order to apply policies, we need to track different flows so that each flow may have many policies applied to it. Once a flow is processed, it will return a match status of Accept, Reject, Drop, No Match, or Not Yet Known.

Once the UML diagram was complete, we transferred it into tables in our database using database conversion techniques. The final database schema result of seven tables that can be seen in Figure 6. We created two tables Policy_clients and Rule_notifications to store the many to many relationships that policies have with clients and rules have with notifications. The other tables consist of policies, clients, rules, notifications, and tags, each of which store information about those objects.

When storing a rule's conditions in the database, we decided to store each condition as its own rule object, with the same action and set of notifications as the condition's rule. Since a rule contains the four possible selectors that will be "ANDED" together, we do not need to record the

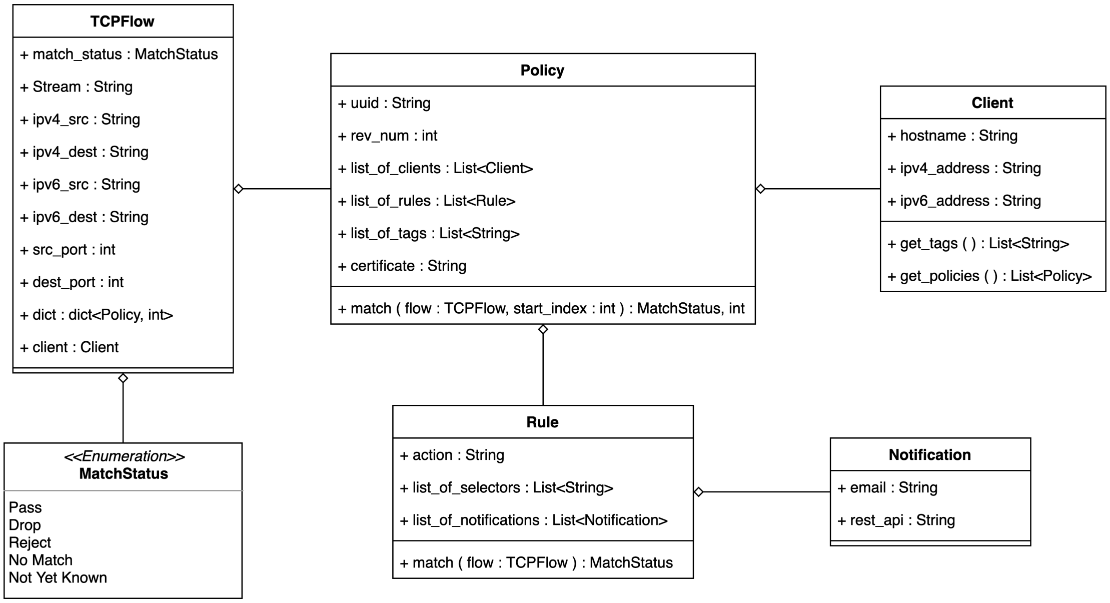


Figure 5: Policy UML

"OR" relationship that conditions inside a rule have. The firewall will apply rules from the database in ascending rule number order. The conditions inside a rule will have increasing rule numbers, but the priority is irrelevant as long as the condition matches than the action and notifications are applied.

5.5 Firewall

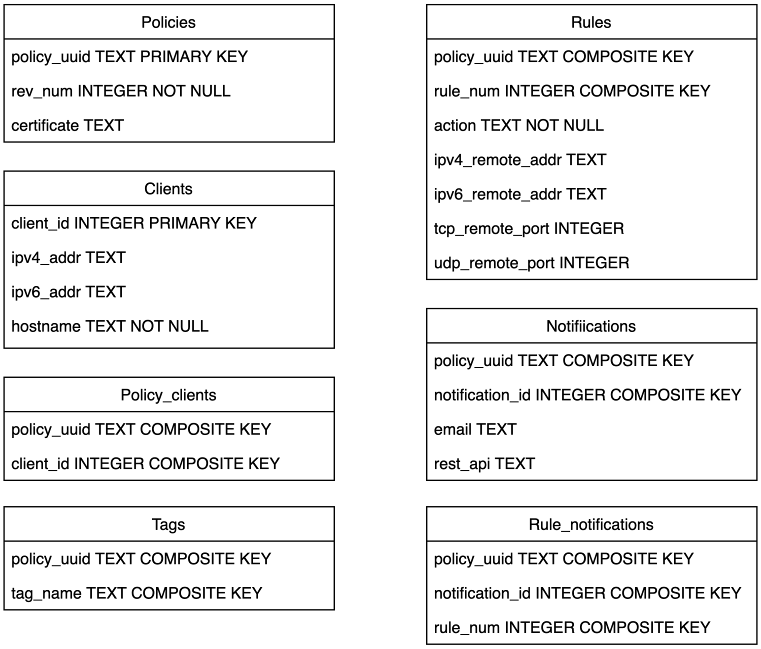
The core of our project is a firewall Rust module that will start different threads to complete the required tasks concurrently.

5.5.1 NFQueue

The middlebox uses the netfilter queue functionality of the Linux kernel for queuing packets that should be inspected by the middlebox or by tshark. Packets that should be inspected are placed in an nfqueue by an iptables rule and the middlebox or tshark then removes the packet from the queue and processes it.

5.5.2 Flows

Each network flow identified by the middlebox is represented as a data structure containing the source and destination IP addresses and port numbers, IP protocol number, HTTP host header if identified by tshark, pointer to a client data structure, the direction of the flow (inbound/outbound), the verdict (pass/drop/reject/not yet determined), cached state used for determining the verdict, and the TCP/UDP stream.



Policies

policy_uuid TEXT PRIMARY KEY
rev_num INTEGER NOT NULL
certificate TEXT

Clients

client_id INTEGER PRIMARY KEY
ipv4_addr TEXT
ipv6_addr TEXT
hostname TEXT NOT NULL

Policy_clients

policy_uuid TEXT COMPOSITE KEY
client_id INTEGER COMPOSITE KEY

Tags

policy_uuid TEXT COMPOSITE KEY
tag_name TEXT COMPOSITE KEY

Rules

policy_uuid TEXT COMPOSITE KEY
rule_num INTEGER COMPOSITE KEY
action TEXT NOT NULL
ipv4_remote_addr TEXT
ipv6_remote_addr TEXT
tcp_remote_port INTEGER
udp_remote_port INTEGER

Notifiications

policy_uuid TEXT COMPOSITE KEY
notification_id INTEGER COMPOSITE KEY
email TEXT
rest_api TEXT

Rule_notifications

policy_uuid TEXT COMPOSITE KEY
notification_id INTEGER COMPOSITE KEY
rule_num INTEGER COMPOSITE KEY

Figure 6: Database Schema

Each time a new packet is received and the verdict has not yet been determined for a flow, the flow is re-evaluated against the policies. Some selectors, such as HTTP host, may sometimes not be able to reach a verdict with only the first packet of a flow but will later reach a verdict upon subsequent packets (pass/drop/reject).

5.5.3 Match Status

When different policies produce different verdicts for the same flow, the most restrictive verdict takes precedence. Within a policy, rules are applied on a first-match basis. Rules are evaluated in order. If a rule returns a verdict or pass, drop, or reject, that immediately becomes the policy's verdict. If a rule does not match the flow, the next rule is evaluated. If the rule does not yet have enough information to make a determination of whether the flow matches that rule, the evaluation of the policy is suspended, the current state is cached, and the policy is not yet able to produce a verdict.

5.5.4 Default Status

When a verdict has not yet been reached for a flow, packets are passed. A verdict will almost always be reached within the first few packets of the flow if not with just the first packet.

5.5.5 Tshark Cooperative Keying

A thread within the middlebox code repeatedly runs tshark by calling the command with the std::process [48] library in Rust. Tshark processes packets waiting in a specified nfqueue and decrypts HTTPS connections using key information provided to it by the middlebox code. Tshark then outputs HTTP host headers along with IP and port information to identify the flow. The middlebox code incorporates this information into its decision-making. The middlebox receives the TLS key information from clients over via UDP.

5.5.6 Centralized Policy Manager

Upon receiving an advertisement from a client, the middlebox uses any tags that the client advertised to retrieve further policies from a central FTP server. These policies are applied in addition to the ones advertised by the client. This allows for centralized management of policies.

5.6 Notification Methods

Notifications within Soteria are triggered when a rule is matched, despite the rule's action. A notification can be sent via a REST-API call or an e-mail, with both supporting either users or manufacturers on the receiving end.

5.6.1 REST-API

The Rust Crate Hyper [49] library was used to make REST-API calls. Hyper is a fast HTTP implementation written for Rust. It allows for high concurrency client and server non-blocking sockets.

Notifying the user is currently done through sending a HTTP POST request to a desired server. This server could be linked to webhooks to allow for notifying the user via their smart home. Additionally, the HTTP POST can be used to notify the user through an app or any other system that can leverage REST API's.

5.6.2 E-mail

The lettre_email [50] library was used to send email notifications in Rust. An e-mail will be built and sent to the provided e-mail address from Soteria's email address. The email will consist of the basic information that the the manufacturer had a matched rule in one of their devices.

6 Results

In this section, we demonstrate Soteria's ability to perform as a protocol based IDS/IPS for network traffic. The system is tested in the following categories: system correctness, throughput performance, and memory/CPU performance. To allow for these tests to be repeatable and done efficiently, we wrote a series of Python scripts. These test scripts were written around the test case specifications outlined in Appendix B.

6.1 System Working Correctly

Automated smoke tests were implemented in Python using the pytest library. Several test cases were written to test various aspects of the middlebox's ability to receive and apply policies. The tests are run on a machine positioned behind the middlebox. The machine on which the tests are executed takes on the role of a client in the tests. Bash scripts were written to provide support to the Python code for performing tasks such as starting and stopping the middlebox code, which require commands to be executed on another machine via SSH.

At the completion of our project, all of the smoke tests pass except the ones that test IPv6. Our test infrastructure is not configured for IPv6, so we marked those two tests as expected to fail.

The tests cover functionality including successfully receiving and applying policies, persisting policies across restarts of the firewall application, retrieving managed policies from the central policy manager, and sending notifications according to policies.

6.2 Throughput Performance

Given that efficiency was a design consideration for this project, we thought it would be prudent to perform throughput testing to see what effect our system had on network flow. Throughput testing on a system is important to see where the system is performing well and where there is room to improve. As such, we drafted a series of experiments that determined just this. These experiments acted as smoke tests, such that any changes we made to the implementation of our system would not make it extremely hard to redo the testing.

To properly see if our system is inhibiting the performance of network flow, we first need to establish a baseline of network performance without it. Since Soteria is run on a Debian Linux VM, IPtables is used for establishing the rules that the firewall enforces. Because the middlebox blocks all traffic on the lab network when Soteria is turned off, we needed a way to allow traffic to flow without the full system running to establish the baseline test.

Utilizing the smoke testing framework established in the system correctness testing, lib.sh was appended with a series of additional functions that would allow for IPtables to be setup for basic network flow and HTTP file sending. The first function added to lib.sh was setup_iptables. This function SSH's into the middlebox and performs the following IPtables commands:

```
$ sudo iptables -F
$ sudo iptables -A POSTROUTING -o eth1 -t nat -j MASQUERADE
$ sudo iptables -A FORWARD -i eth0 -t filter -j ACCEPT
```

The first command deletes all existing IPTables rules so that if any rules were leftover from a previous config they will not interfere with the existing run. The next command appends POSTROUTING to the chain and sets the output interface to eth1, finally setting the table to nat and the target to MASQUERADE. The final command appends FORWARDing to the input of eth0, setting the table to filter and the target to accept. These commands are important as they allow the client to access devices external to the network as if there is not a firewall present.

Additionally, to allow for easy teardown, teardown_iptables was also added to lib.sh. This performs the following command on the middlebox until there are no IPTables rules resent.

```
$ sudo iptables -F
```

For the tests to follow, in addition to the IPTables setup and teardown was outlined in this section and the starting and stopping of the firewall was outlined in the system correctness tests, the tests needed a way to indicate performance metrics. As such, we decided to use three different metrics. The first of which is a large file transfer over HTTP. This large file transfer allows for us as a team to see if Soteria adds a noticeable discrepancy in network flow. As such, two additional methods were outlined in lib.sh. The first is send_file_to_client method. This method first SSH's a machine external to the lab subnet and performs the following commands:

```
$ sudo cd /root/iot-security-mqp/tests
$ python3 HTTP_transfer.py -m send -i test.txt
```

This command changes the directory to the testing folder and runs a Python file made specifically for these tests. This Python file has two flags that are appended. The first is "-m" followed by the mode of operation. The two acceptable modes are sending and receiving. The next flag is -i followed by the input. For sending, the input is the file name. For receiving, the input is the IP address/hostname. As this method is designed to send the file to the client, the file name is large.m4v and the ip address is 192.168.7.237 respectively.

The next test that will be run on both the base-case and firewall is a ping test. This test will perform the following command for an elapsed time of ten minutes from the Client VM:

```
$ ping ipv4.google.com
```

The time frame of ten minutes was chosen such that approximately 600 packets will be sent and received. ipv4.google.com was chosen so that it would direct to the IPv4 specific google.com domain and would ensure some sort of stability in the testing, as google.com is a large site with theoretically reliable uptime.

The final test to be run is Speedtest CLI. Speedtest CLI is a command line interface for speedtest.net that allows a user to get the speedtest analytics without needing a graphical user interface.

These three tests done in tandem should allow us as a research team to see if Soteria adds a noticeable overhead to network flow.

6.2.1 Baseline Tests

The baseline test 3-1 measures the throughput of the network without the team's code present. before the test was run, the middlebox was setup for IPTables forwarding without any filtering enabled. The test first started a timer. After the timer is started, a 4GB file was downloaded to the client through the middlebox from a server not on the subnet but still on the LAN. Once the file transfer is complete, the client then uploaded the same 4GB file to the server through the middlebox. Finally, the time elapsed in seconds is displayed. After running the test, the elapsed time was:

```
root@mqp-server:~/iot-security-mqp/tests# cd /root/iot-security-mqp/
tests && python3 HTTP_transfer.py -m send -i large.m4v
the mode is: send
the input is: large.m4v
Server listening....
Got connection from ('192.168.7.4', 50596)
Server received b'connected'
Done sending
test elapsed time:  177.90483601400047
```

This elapsed time indicates a baseline for test 3-4 to be compared to.

Baseline test 3-2 measures the Ping performance without our code. Before the test was run, the middlebox was setup for IPTables forwarding without any filtering enabled. Additionally, this test was run very early in the morning to ensure that minimal devices were on the network and using bandwidth. The test first started with pinging ipv4.google.com from the client for 10 minutes. These 10 minutes were done from a stopwatch. After the 10 minutes were elapsed, the ping was terminated and the results were displayed. The ping results for the middlebox without the team's code are displayed below:

```
(client_env) root@mqp-client:/srv/iot-security-mqp/tests# ping
ipv4.google.com
PING ipv4.l.google.com (172.217.4.78) 56(84) bytes of data.
64 bytes from ord37s18-in-f14.1e100.net (172.217.4.78): icmp_seq=1
ttl=50 time=35.6 ms
64 bytes from ord37s18-in-f14.1e100.net (172.217.4.78): icmp_seq=2
ttl=50 time=40.9 ms
64 bytes from ord37s18-in-f14.1e100.net (172.217.4.78): icmp_seq=3
ttl=50 time=35.9 ms
...
64 bytes from ord37s18-in-f14.1e100.net (172.217.4.78): icmp_seq=600
ttl=50 time=36.1 ms
^C
--- ipv4.l.google.com ping statistics ---
600 packets transmitted, 599 received, 0.166667% packet loss, time
1485ms rtt min/avg/max/mdev = 34.346/36.298/50.324/2.092 ms
```

These IPv4 ping statistics will be used as a baseline when compared with Firewall test 3-5.

Finally, Baseline test 3-3 measured the Speedtest CLI without our code. Before the test was run, the middlebox was setup for IPTables forwarding without any filtering enabled. Additionally, this test was also run very early in the morning to ensure that minimal devices were on the network and using bandwidth. The test first started with running the speedtest command. Once the command was done running, the results were shown. The Speedtest CLI results for the middlebox without the team's code are displayed below:

```
(client_env) root@mqp-client:/srv/iot-security-mqp/tests# speedtest

   Speedtest by Ookla

     Server: Spectrum - Oxford, MA (id = 2405)
        ISP: Spectrum
    Latency:     8.23 ms   (0.41 ms jitter)
   Download:   457.89 Mbps (data used: 494.0 MB)
     Upload:    18.53 Mbps (data used: 20.2 MB)
Packet Loss:     0.0%
 Result URL: https://www.speedtest.net/result/c/e8a63a70-2b81-4df3
 -bbf7-a0cb69adaf6b
```

The Speedtest CLI results will be used as a baseline for comparison against Firewall test 3-6.

6.2.2 Firewall Tests

The Firewall test 3-4 measures the throughput of the network with the team's firewall code present. Before the test was run, the middlebox was setup to run the firewall code. The test first started a timer. After the timer is started, a 4GB file was downloaded to the client through the middlebox from a server not on the subnet but still on the LAN. Once this was done, the client then uploaded the same 4GB file to server through the middlebox. Finally, the time elapsed in seconds is displayed. After running the test, the elapsed time for downloading to the client was:

```
root@mqp-server:~/iot-security-mqp/tests# cd /root/iot-security-mqp/
tests && python3 HTTP_transfer.py -m send -i large.m4once this was done
v
the mode is: send
the input is: large.m4v
Server listening....
Got connection from ('192.168.7.4', 50588)
Server received b'connected'
Done sending
test elapsed time:  186.73574227299832
and the elapsed time for uploading to the server was:
```

Compared to Baseline test 3-1's elapsed time of 177.90:48 seconds, an increase of 8.83 seconds with the addition of the team's firewall results in a 4.84% difference between when the firewall is running and when it is not running. This result indicates that the presence of the team's firewall code does not add a dramatic impact on network performance, as roughly a 5% increase in data transfer time is with a reasonable limit given the network filtering that the firewall performs.

Firewall test 3-5 measures the Ping performance without the team's firewall code present. Before the test was run, the middlebox was setup to run the firewall code. Additionally, this test was run very early in the morning to ensure that minimal devices were on the network and using bandwidth. The test first started with pinging ipv4.google.com from the client for 10 minutes. These 10 minutes were done from a stopwatch. After the 10 minutes were elapsed, the ping was be terminated and the results were displayed. The ping results for the middlebox with the team's code are displayed below:

```
(client_env) root@mqp-client:/srv/iot-security-mqp/tests# ping
ipv4.google.com
PING ipv4.l.google.com (172.217.6.14) 56(84) bytes of data.
64 bytes from ord38s01-in-f14.1e100.net (172.217.6.14): icmp_seq=1
ttl=50 time=37.4 ms
64 bytes from ord38s01-in-f14.1e100.net (172.217.6.14): icmp_seq=2
ttl=50 time=36.0 ms
64 bytes from ord38s01-in-f14.1e100.net (172.217.6.14): icmp_seq=3
ttl=50 time=35.1 ms
...
64 bytes from ord38s01-in-f14.1e100.net (172.217.6.14): icmp_seq=599
ttl=50 time=39.6 ms
^C
--- ipv4.l.google.com ping statistics ---
599 packets transmitted, 599 received, 0% packet loss, time
1456ms rtt min/avg/max/mdev = 34.258/36.119/51.337/2.231 ms
```

Compared to ping results of Baseline test 3-2, Firewall test 3-5 indicates that the presence of the firewall again does not hinder the network throughput in a significant way. While Baseline test 3-2 had a 0.17% packet loss and Firewall test 3-5 exhibited no packet loss, The packet loss from Baseline test 3-2 is not a significant metric as this indicates the 1 packet that was lost over 600 packets sent in total was an outlier.

Finally, Firewall test 3-6 measured the Speedtest CLI with the team's firewall code present. Before the test was run, the middlebox was setup to run the firewall code. Additionally, this test was also run very early in the morning to ensure that minimal devices were on the network and using bandwidth. The test first started with running the speedtest command. Once the command was done running, the results were shown. The Speedtest CLI results for the middlebox with the team's code are displayed below:

```
(client_env) root@mqp-client:/srv/iot-security-mqp/tests# speedtest
```

```
   Speedtest by Ookla

     Server: Spectrum - Oxford, MA (id = 2405)
        ISP: Spectrum
    Latency:     9.27 ms   (0.69 ms jitter)
   Download:   398.57 Mbps (data used: 522.7 MB)
     Upload:    18.93 Mbps (data used: 23.2 MB)
Packet Loss:     0.0%
 Result URL: https://www.speedtest.net/result/c/2b0c6ccd-cded-4ab7
 -88c4-5c4074046b23
```

When compared to Baseline test 3-3, Firewall test 3-4 indicates an 11.89% increase in latency, 13.85% decrease in download speed, 7.28% increase in upload speed and a 0% difference in packet loss. While the increase in latency and decrease in download speed with the addition of the firewall may indicate that the firewall is adding a notable amount of latency, the 7.28% increase in upload speed and the results of the prior tests indicate that the perceived throughout degradation may not be the case. In addition to that, on the lab network used, the maximum download speed that was paid for at the time of running these tests was 400 Mbps. Given the download speed of 457.89 Mbps from Baseline test 3-3 is 13.50% more then the network speed being paid for, we as a team would like to indicate this to the variability of spectrum's data rates. If the download were capped at paid for 400 Mbps, then the download speed decrease with the presence of the firewall would be 0.36%, which is a reasonable band of tolerance, indicating again that the presence of the firewall does not add a notable degradation in network throughput.

6.3 Memory/CPU Performance

Given that Soteria is designed to be run on a router, it is worthwhile to test the memory and CPU performance of our system. Routers are typically lightweight systems, therefore Soteria should not be consuming too many resources. While the Rust programming language does have the same benefits of C in terms of speed performance, it is much worse in terms of memory usage.

The memory tests we ran on our middlebox are from the `top` command in the terminal. The columns of this data that we used are DATE, VIRT, RES, SHR, CPU, and MEM. We used these columns to measure the timestamp, total amount of memory consumed by a process, the memory consumed by a process in RAM, the amount of memory shared with other processes, the CPU usage, and the percentage of RAM available respectively. The script to generate a CSV file with this data can be seen below.

```
FILENAME="$1"
rm -rf $FILENAME
> $FILENAME
echo "writing to $FILENAME"
echo "TIME_STAMP, VIRT, RES, SHR, %CPU, %MEM" | tee -a $FILENAME
```

```
while :
do
    DATE=‘date +"%H:%M:%S:%s%:z"‘
    echo -n "$DATE, " | tee -a $FILENAME
    # break apart the info from the top command
    LINE=‘top -b -n 1 | grep -w python | tr -s ’ ’‘
    # VIRT is total amount of memory consumed by a process
    VIRT=‘echo $LINE | cut -d ’ ’ -f 5‘
    echo -n "$VIRT, " | tee -a $FILENAME
    # RES is the memory consumed by a process in RAM
    RES=‘echo $LINE | cut -d ’ ’ -f 6‘
    echo -n "$RES, " | tee -a $FILENAME
    # SHR is the amount of memory shared with other processes
    SHR=‘echo $LINE | cut -d ’ ’ -f 7‘
    echo -n "$SHR, " | tee -a $FILENAME
    # CPU usage
    CPU=‘echo $LINE | cut -d ’ ’ -f 9‘
    echo -n "$CPU, " | tee -a $FILENAME
    # Expresses RES value as a percentage of the total RAM available
    MEM=‘echo $LINE | cut -d ’ ’ -f 10‘
    echo "$MEM" | tee -a $FILENAME
    sleep 1
done
```

The first condition we tested these five metrics across was the firewall running on the router with no traffic or policies. This test gives a baseline of how much memory and CPU our firewall process consumes.

- The total memory consumed by the middlebox process was 359,348k.
- The memory consumed by the process in RAM was 8,984k.
- The memory shared between processes was 7,616k.
- The CPU usage ranged from 100% to 200% on two cores, averaging around 165%.
- The percentage of RAM consumed was 20%.

The second condition we tested was the firewall running with network traffic. This test gives us a baseline of how our firewall would normally be running.

- The total memory consumed by the middlebox process remained at 359,348k.
- The memory consumed by the process in RAM increased to 11,648k.
- The memory shared between processes increased to 10,000k.

- The CPU usage ranged from 130% to 200% on two cores, averaging around 175%.
- The percentage of RAM consumed increased to 30%.

The last is the firewall when a policy is applied to the system. This test gives us a baseline of how our firewall would handle updates to the database.

- The total memory consumed by the middlebox process remained at 359,348k.
- The memory consumed by the process in RAM increased to 11,648k.
- The memory shared between processes increased to 10,000k.
- The CPU usage ranged from 130% to 200% on two cores, averaging around 175%.
- The percentage of RAM consumed increased to 30%.

Overall we can see that the memory and CPU performance of Rust can get rather large. This is typical in Rust do to the lack of shared memory for security reasons. By creating more data objects, Rust is a more secure language at the cost of memory performance. Therefore our solution may require a heavy duty router with a decent amount of memory and RAM. Future tests can include the network traffic before and after a policy is sent, as well as test the size of a large policy on the system.

7 Discussion

This section reflects upon Soteria's ability and performance as well as the experience we have gained throughout working on this project. First, we analyze the results and what they indicate about the success of both our goals and research question. Then, we discuss some of the engineering difficulties that limited both the scope and feature set of our project. Finally, we look into future development considerations where we outline possible directions of further engineering and research for any onlooking research individuals or teams.

7.1 Engineering Lessons Learned

Throughout the time we have spent on this project, we have gained more insight about the system and community we are developing for. The lessons outlined below are ones that we had no awareness of when beginning our project and may be valuable to detail for future parties who will pursue this project or related work.

7.1.1 Shared Memory

Initially, the middlebox code was designed without an understanding of the differences between the multi-threading paradigms of C and Rust. In C, on which Rust is based, multi-threaded programs share information between threads through shared memory. In Rust, that is considered unsafe and is not allowed. In Rust, multi-threaded programs share information between threads by sending explicit messages between threads; for example, if thread A wants to share a data structure with thread B, it must send the object in a message to thread B, and thread B must explicitly check for, receive, and process the message. This is different than multi-threading in C, where thread A would simply modify memory shared with thread B and locks would be employed to prevent race conditions. We initially engineered the middlebox code without understanding that Rust does not allow sharing memory between threads in the manner that we intended. When ran into this issue when adding the tshark thread to the code and had to perform a major refactor.

Working on this project flipped our existing notions of how multi-threading programs are structured and exposed us to new ideas that violated principles we thought were universally true. Our existing education and experience had not exposed us to multi-threading paradigms other than the traditional one of C.

7.1.2 Libraries

When developing, we ran into unexpected issues with certain libraries. These libraries either had an unclear documentation or issues with another library.

7.1.2.1 OpenSSL

OpenSSL is an opensource SSL cryptographic library [51] that is leveraged heavily throughout this MQP. During the normal development phase, it was used for establishing TLS keys and later used for the implementation of TLS session key sharing. As Rust was made within the past decade, its insurgence has brought a fair share of benefits, but there is nevertheless opportunity for improvement. Key libraries need to be ported to Rust and the Rust community is smaller than that of the Python, Java, or C community. It takes time for any given library to be supported, or, in our case, updated accordingly. To help mitigate this process, Rust keeps a Cargo.toml and a Cargo.lock file. The Cargo.toml file is where the developer puts the version of what packages they want. These packages gets interpreted to the Cargo.lock file at compile time. As such, any edits made to the Cargo.toml will be reflected in the Cargo.lock, which will then download the appropriate packages and inject them into the binary.

Because the Rust community is small, as stated earlier, package updates tend to happen somewhat sporadically. In our case, with OpenSSL [51], up until recently, 0.9.23 was passing the build, but anything following it was not. When we started the implementation phase of Soteria, OpenSSL was at 0.9.48, but was only building at 0.9.23. As such, when we performed:

```
$ cargo run
```

the system would work as expected, but when we performed:

```
$ cargo build
```

and we were greeted with a compatibility error. This was encountered rarely throughout the project development but began to increase in frequency as we began looking into cross-compatibility. At the time of writing this paper, the current version of OpenSSL, 0.9.54, is passing the build.

This project again has taught us valuable lessons about programming, this time about working with packages that are newly developed, with little documentation. While performance and stability guarantees are enticing when starting a large project, it is important to remember that it takes time to update these packages and their compatibility. As OpenSSL and Rust are open-sourced, these efforts are often community-driven and require those that want the package to work to be the ones to write the necessary code for the community. Beyond that, it pays to read a package's release notes and documentation. Doing so will ensure that the packages implemented will not cause problems later on in development.

7.1.3 Cross-Compilation

We as a team wanted to make our research as accessible to as many users and manufacturers as possible. Given this goal, compiling our codebase down to fit on a router was important to us as a team. When we ready to look into cross-compilation, again, we were reminded of the value of the motto of WPI: Leer Un Kunst. There is a reason why theory comes before practice in the motto. Without the theory, there is nothing to practice. Performing proper research into the task at hand is invaluable. Doing so will allow the "practice", or engineering efforts to run a lot smoother and ultimately in a quality final product.

A decent amount of the Rust community works with cross-compilation; however, their efforts are dispersed among various projects. As such, Rustaceans (the self-dubbed name for the Rust development community) at times are left in forums for years waiting for support. We once again attribute this to the size and dispersion of the Rustacean community.

7.1.3.1 Rustup

Rustup [52] is the rust install and version management tool. Like Cargo, Rustup is responsible for managing a variety of tasks that make the rust development process a lot smoother. Rustup is also the toolchain manager for rust. Whenever anything is built in rust, it is compiled down to a binary executable. For this compilation to be possible, a toolchain is required. Out of the box, rust supports a fair number of toolchains. When referring to a toolchain, it is referred to by its host triple. The following command will show what toolchains are installed on your machine:

```
$ rustup show
```

Run on a modern macOS system, this command will likely show:

```
Default host: x86_64-apple-darwin
rustup home:  /Users/<username>/.rustup

installed targets for active toolchain
--------------------------------------



x86_64-apple-darwin

active toolchain
----------------

stable-x86_64-apple-darwin (default)
rustc 1.41.0 (5e1a79984 2020-01-27)
```

where the compile triple is "x86_64-apple-darwin". This triple means that the system is running x86_64 assembly under the hood, is made by Apple and is running the Darwin operating system. For the TP-Link Archer C7 Routers provided, the compile triple was "mips-openwrt-linux-gcc". As stated earlier, each triple is a toolchain. As this toolchain was not one included by default as a downloadable toolchain within the Rustup CLI, it had to be imported. in addition to this, when a toolchain is imported, it does not guarantee that the binary will compile down to work on that toolchain, as compatibility issues are always present with niché applications. When the OpenWRT toolchain for the MIPS architecture was imported, this is exactly what happened. We attribute this either to linking issues with Rustup or issues within the OpenWRT package. The result of this was to move towards utilizing a more generic compile triple in the hopes that it would still run as long as the generic compile triple utilized the same assembly architecture. As such, the resulting compile triple was "mips-unknown-linux-musl" which was supposedly supported by Rustup. However, upon further investigation outside of the documentation and into the forums and Github Issues of Rust and Rustup indicated that there was an issue with MIPS compilation with Rust that has been present in the community for a while.

7.2 Future Development Considerations

During development, we realized that the scope of this project was greater than the allotted time we had to work on it. As such, we have provided a list of features that we were unable to implement, which we believe to expand on the project idea:

1. Increase the amount and variety of selectors used for matching policies. Examples include autonomous systems, TLS CA's, and HTTP headers.

2. Implement filtering on the HTTP host headers extracted from cooperative keying.

3. Verify advertisement certificate with certificate authority.

4. Send email notifications with information on which rule from which policy was matched.

5. Implement a mobile application to aid in user notifications.

8 Conclusion

Manufacturer collaboration with firewall policies is an underdeveloped field. Our protocol IDS/IPS, Soteria, makes the assumption of manufacturer collaboration in a plug-and-play security model. With this assumption we are able to achieve a 100% accuracy in properly filtering network traffic for IoT devices on a residential network. Our firewall only allows IoT devices to communicate with the devices that the manufacturer or third party allows, applying the more strict policy if several rules match the traffic flow. By reducing the communication of IoT devices to only the desired sources and destinations, the attack vectors of a bad actor are also reduced. When an IoT device is compromised, the bad actor is unable to freely open network connections and may only communicate with the current network traffic rules or DoS the device off the network. This prevents bot managers from connecting their bots to a command and control server on an unfamiliar destination in the firewall policies. This satisfies our initial goal of seeing how effective manufacturer support is in a plug-and-play security model. A 100% accuracy of network filtering can be achieved with around a 5% increase in throughput on a router with the memory used by the middlebox process being 359,348 kilobytes of RAM. Additionally, TLS inspection allows our firewall to inspect REST API calls over HTTPS.

Soteria's policies also include the ability to notify a user or manufacturer if a device reaches a firewall rule that it should not have. With a notification, a user is able to take remedial action on the compromised device. A manufacturer will have an increased awareness of their products vulnerabilities and be able to patch their firmware accordingly.

This project includes automated Python tests for system correctness, throughput performance, and memory performance. We have tested our system with three virtual machines on a secure network. We failed to add a wireless access point to test IoT devices on a residential network due to Rust cross-compilation errors. Further testing of this system could be done with IoT devices on a residential network, or by opening it to the public to be hacked.

References

[1] D. Goodin, "Internet-paralyzing Mirai botnet comes roaring back with new strain," 11/29/ 2017. [Online]. Available: https://arstechnica.com/information-technology/2017/11/internet-paralyzing-mirai-botnet-comes-roaring-back-with-new-strain/

[2] M. Starr, "Fridge caught sending spam emails in botnet attack," -01-19 2014. [Online]. Available: https://www.cnet.com/news/fridge-caught-sending-spam-emails-in-botnet-attack/

[3] R. Crist, "New study details a security flaw with Philips Hue smart bulbs," 11/03/ 2016. [Online]. Available: https://www.cnet.com/news/new-study-details-a-security-flaw-with-philips-hue-smart-bulbs/

[4] L. Constantin, "Update your Belkin WeMo devices before they become botnet zombies," 11/07/ 2016. [Online]. Available: https://www.computerworld.com/article/3138991/update-your-belkin-wemo-devices-before-they-become-botnet-zombies.html

[5] C. Talos, "Vulnerability spotlight: Multiple vulnerabilities in Samsung SmartThings Hub," 07/26/ 2018. [Online]. Available: https://blog.talosintelligence.com/2018/07/samsung-smartthings-vulns.html

[6] B. Schneier, "The Internet of Things is wildly insecure — and often unpatchable," -01-06 2014. [Online]. Available: https://www.wired.com/2014/01/theres-no-good-way-to-patch-the-internet-of-things-and-thats-a-huge-problem/

[7] W. Witkowski, "Intel admits vulnerability, but plays down effects; stock slides, amd gains," Jan 4, 2018. [Online]. Available: https://www.marketwatch.com/story/intel-stock-headed-for-worst-day-in-more-than-a-year-amd-pops-on-chip-design-flaw-report-2018-01-03

[8] "Openwrt wireless freedom." [Online]. Available: https://openwrt.org/

[9] "Replay attack." [Online]. Available: https://en.wikipedia.org/wiki/Replay_attack

[10] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A. Sadeghi, and S. Tarkoma, "IoT SENTINEL: Automated device-type identification for security enforcement in IoT," June 2017, pp. 2177–2184.

[11] B. Bezawada, M. Bachani, J. Peterson, H. Shirazi, I. Ray, and I. Ray, "Iotsense: Behavioral fingerprinting of iot devices," *arXiv.org*, 2018, iD: proquest2071992576.

[12] D. Chen, N. Zhang, Z. Qin, X. Mao, Z. Qin, X. Shen, and X. Li, "S2m: A lightweight acoustic fingerprints-based wireless device authentication protocol," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 88–100, February 2017.

[13] M. Nobakht, V. Sivaraman, and R. Boreli, "A host-based intrusion detection and mitigation framework for smart home IoT using OpenFlow," pp. 147–156, 2016.

[14] T. Golomb, Y. Mirsky, and Y. Elovici, “Ciota: Collaborative iot anomaly detection via blockchain,” Mar 10, 2018. [Online]. Available: https://www.openaire.eu/search/publication?articleId=od________18::4583c50b182dbb520c662f7a66e10d66

[15] N. Apthorpe, D. Reisman, S. Sundaresan, A. Narayanan, and N. Feamster, “Spying on the smart home: Privacy attacks and defenses on encrypted IoT traffic,” Aug 16, 2017. [Online]. Available: \url{https://www.openaire.eu/search/publication?articleId=od________18::f1b99748dbf3e8dd55c8208dd109d2e6}

[16] E. J. Cho, J. H. Kim, and C. S. Hong, “Attack model and detection scheme for botnet on 6lowpan,” pp. 515–518, 2009, iD: scopus2-s2.0-70350440685.

[17] L. Liu, B. Xu, X. Zhang, and X. Wu, “An intrusion detection method for internet of things based on suppressed fuzzy clustering,” *EURASIP Journal on Wireless Communications and Networking*, vol. 2018, no. 1, pp. 1–7, 2018.

[18] I. E. T. Force, “TOTP: Time-based one-time password algorithm,” 2011. [Online]. Available: https://tools.ietf.org/html/rfc6238

[19] M. Data, “The defense against ARP spoofing attack using semi-static ARP cache table,” pp. 206–210, 2018.

[20] N. Feamster, “Outsourcing home network security,” pp. 37–42, 2010.

[21] C. R. Taylor, C. A. Shue, and M. E. Najd, “Whole home proxies: Bringing enterprise-grade security to residential networks,” p. ¡xocs:first xmlns:xocs=””/¿, 2016.

[22] C. R. Taylor and C. A. Shue, “Validating security protocols with cloud-based middleboxes,” pp. 261–269, 2016.

[23] M. Gajewski, J. M. Batalla, G. Mastorakis, and C. X. Mavromoustakis, “A distributed ids architecture model for smart home systems,” *Cluster Computing*, vol. 22, 2017.

[24] V. Sivaraman, H. H. Gharakheili, A. Vishwanath, R. Boreli, and O. Mehani, “Network-level security and privacy control for smart-home IoT devices,” pp. 163–167, 2015.

[25] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, “Making middleboxes someone else’s problem: Network processing as a cloud service,” *ACM SIGCOMM Computer Communication Review*, vol. 42, 10 2012.

[26] “Blue Cave — networking.” [Online]. Available: https://www.asus.com/us/Networking/Blue-Cave/

[27] “Bitdefender BOX - IoT security solution for all connected devices.” [Online]. Available: https://www.bitdefender.com/box/v2/

[28] P. Miller, “Dojo is another oddly shaped solution to securing your home network,” -05-31T16:53:27-04:00 2017. [Online]. Available: https://www.theverge.com/circuitbreaker/2017/5/31/15721434/bullguard-dojo-internet-of-things-network-security-router

[29] "F-Secure SENSE — secure router and app." [Online]. Available: https://www.f-secure.com/en_US/web/home_us/sense

[30] "Gryphon mesh Wifi security router and parental control system." [Online]. Available: https://gryphonconnect.com/

[31] "Norton Core router — secure wifi router." [Online]. Available: https://us.norton.com/core

[32] T. L. V. Sperling, F. L. de Caldas Filho, R. T. de Sousa, E. L. M. C. Martins, and R. L. Rocha, "Tracking intruders in IoT networks by means of dns traffic analysis," pp. 1–4, 2017.

[33] H. Sedjelmaci, S. M. Senouci, and M. Al-Bahri, "A lightweight anomaly detection technique for low-resource iot devices: A game-theoretic methodology," p. ¡xocs:first xmlns:xocs=""/¿, 2016, iD: scopus2-s2.0-84981332564.

[34] M. Bierma, A. Brown, T. Delano, T. M. Kroeger, and H. Poston, "Locally operated cooperative key sharing (LOCKS)," pp. 356–362, 2017.

[35] S. Hachana, N. Cuppens-Boulahia, and F. Cuppens, "Mining a high level access control policy in a network with multiple firewalls," *Journal of Information Security and Applications*, vol. 20, pp. 61–73, 2015.

[36] "Python." [Online]. Available: https://www.python.org/

[37] "Tox." [Online]. Available: https://tox.readthedocs.io/en/latest/

[38] "Pylint." [Online]. Available: https://www.pylint.org/

[39] "Flake8." [Online]. Available: https://flake8.pycqa.org/en/latest/

[40] M. Team, "Why rust for safe systems programming," July 22, 2019. [Online]. Available: https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/

[41] Y. Lui, "Floodlight." [Online]. Available: https://web.cs.wpi.edu/~yliu25/floodlightsetup.html

[42] "Openwrt project: Table of hardware." [Online]. Available: https://openwrt.org/toh/start

[43] "Openwrt project: Dumb access point / dumb ap." [Online]. Available: https://openwrt.org/docs/guide-user/network/wifi/dumbap

[44] RazrFalcon, "roxmltree." [Online]. Available: https://github.com/RazrFalcon/roxmltree

[45] "Sqlite." [Online]. Available: https://www.sqlite.org/index.html

[46] "Rust sqlite." [Online]. Available: https://rust-lang-nursery.github.io/rust-cookbook/database/sqlite.html

[47] "Diesel." [Online]. Available: http://diesel.rs/

[48] "std::process::command." [Online]. Available: https://doc.rust-lang.org/std/process/struct.Command.html

[49] “hyper.rs — hyper.” [Online]. Available: https://hyper.rs/

[50] “lettre_email.” [Online]. Available: https://docs.rs/lettre_email/0.9.2/lettre_email/

[51] “openssl - rust.” [Online]. Available: https://docs.rs/openssl/0.10.28/openssl/

[52] “Getting started.” [Online]. Available: https://www.rust-lang.org/learn/get-started

A Policy XSD Schema

Listing 1: Schema for advertisement

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="firewallAction">
    <xs:restriction base="xs:string">
      <xs:enumeration value="pass"/>
      <xs:enumeration value="drop"/>
      <xs:enumeration value="reject"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="notification">
    <xs:sequence>
      <xs:element name="email" type="xs:string"
          maxOccurs="unbounded" minOccurs="0"/>
      <xs:element name="rest-api" type="xs:string"
          maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="rule">
    <xs:sequence>
      <xs:element name="condition" maxOccurs="unbounded" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
                  <xs:element name="ipv4-remote-addr" type="xs:integer"
                    maxOccurs="unbounded" minOccurs="0"/>
                        <xs:element name="ipv6-remote-addr" type="xs:integer"
                      maxOccurs="unbounded" minOccurs="0"/>
                        <xs:element name="tcp-remote-port" type="xs:integer"
                  maxOccurs="unbounded"
                        <xs:element name="udp-remote-port" type="xs:integer"
                  maxOccurs="unbounded" minOccurs="0"/>
                        <xs:element name="autonomous-system" type="xs:integer"
                maxOccurs="unbounded" minOccurs="0"/>
                        <xs:element name="http-host" type="xs:string"
        maxOccurs="unbounded" minOccurs="0"/>
            <xs:element name="tls-ca" type="xs:string"
        maxOccurs="unbounded" minOccurs="0"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="notification" type="notification"
     maxOccurs="unbounded" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="action" type="firewallAction"/>
 </xs:complexType>

 <xs:element name="policy">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="tag" type="xs:string"
      maxOccurs="unbounded" minOccurs="0"/>
    <xs:element name="rule" type="rule"
      maxOccurs="unbounded" minOccurs="0"/>
    <xs:element name="ca" type="xs:string"
      maxOccurs="1" minOccurs="1"/>
   </xs:sequence>
   <xs:attribute name="revision" type="xs:integer"/>
  </xs:complexType>
 </xs:element>

</xs:schema>
```

B Test Cases

B.1 System Working Correctly

Test Number: 1-1
Test Name: Firewall drops packet based on a policy
Precondition: Middlebox has a clean database and does not know about client.
Test Steps:

1. Client sends advertisement with a policy that blocks 1.1.1.1.
2. Client sends DNS request to 1.1.1.1.

Pass/fail Criteria:

1. DNS request should be dropped.

Test Number: 1-2
Test Name: Firewall passes packet based on a policy
Precondition: Middlebox has a clean database and does not know about client.
Test Steps:

1. Client sends advertisement with a policy that passes 1.1.1.1.
2. Client sends DNS request to 1.1.1.1.

Pass/fail Criteria:

1. DNS request should be passed.

Test Number: 1-3
Test Name: Firewall passes unknown client traffic
Precondition: Middlebox has a clean database and does not know about client.
Test Steps:

1. Client sends DNS request to 1.1.1.1.

Pass/fail Criteria:

1. DNS request should be passed.

Test Number: 1-4
Test Name: Firewall drops known client traffic
Precondition: Middlebox has a clean database and does not know about client.
Test Steps:

1. Client sends advertisement with a policy that passes 1.1.1.2.

2. Client sends DNS request to 1.1.1.1.

Pass/fail Criteria:

1. DNS request should be dropped.

Test Number: 1-5
Test Name: Firewall applies both selectors inside a condition in a policy
Precondition: Middlebox has a clean database and does not know about client.
Test Steps:

1. Client sends advertisement with a policy condition that passes only if a flow has IPv4 address 1.1.1.1 and UDP port 53.

2. Client sends DNS request to 1.1.1.1 over UDP port 53.

Pass/fail Criteria:

1. DNS request should be passed.

Test Number: 1-6
Test Name: Firewall applies either condition inside a rule in a policy
Precondition: Middlebox has a clean database and does not know about client.
Test Steps:

1. Client sends advertisement with a policy rule that passes with either the condition of a flow has IPv4 address 1.1.1.1 or the condition that a flow has UDP port 53.

2. Client sends DNS request over UDP port 53.

Pass/fail Criteria:

1. DNS request should be passed.

Test Number: 1-7
Test Name: Firewall tests each selector.
Precondition: Middlebox has a clean database and does not know about client.
Test Steps:

1. Client sends advertisement with a policy rule that passes with the conditions that a flow has IPv4 address 1.1.1.1, that a flow has IPv6 address 1:1:1:1:1:1:1:1, that a flow has UDP port 53, or that a flow has TCP port 80.

2. Client sends DNS request to 1.1.1.1.

3. Client sends DNS request to 1:1:1:1:1:1:1:1.

4. Client sends DNS request over UDP port 53.

5. Client sends HTTP request over TCP port 80.

Pass/fail Criteria:

1. All four requests should be passed.

Test Number: 1-8
Test Name: Firewall tests each notification.
Precondition: Middlebox has a clean database and does not know about client.
Test Steps:

1. Client sends advertisement with a policy rule that drops with the conditions that a flow has IPv4 address 1.1.1.1. This condition also contains an email notification to mqpiotnet@gmail.com and a REST API call to https://www.smtpeter.com/v1/send?access_token = {YOUR_API_TOKEN} where YOUR_API_TOKEN has sender and reciever mqpiotnet@gmail.com.

2. Client sends DNS request to 1.1.1.1.

Pass/fail Criteria:

1. A notification email should appear in mqpiotnet@gmail.com inbox from the middlebox.

2. The REST API call should be made to https://www.smtpeter.com/v1/send?access_token = {YOUR_API_TOKEN} which will also send an email to the same mqpiotnet@gmail.com inbox.

Test Number: 1-9
Test Name: Firewalling based on tags
Precondition: There are two clients advertising policies, A and B. Each client has a tag in its policy. Client A's policy says to allow traffic to the tag contained in the client B's policy.
Test Steps:

1. The two clients advertise their policies.

2. The middlebox receives the policies.

3. Client A sends traffic to client B.

4. Client B sends traffic to client A.

Pass/fail Criteria: Client A's traffic reaches client B but client B's traffic to client A is dropped.

Test Number: 1-10
Test Name: Python Testing
Precondition: Client code is not running
Test Steps:

1. Tester runs client code

2. Client code executes properly

3. Client code is terminated

Pass/fail Criteria:

1. Python code is properly threaded
2. Threads close correctly

Test Number: 1-11
Test Name:
Precondition:
Test Steps:

1.

Pass/fail Criteria:

1.

Test Number: 1-12a
Test Name: Middlebox retrieves managed policy based on tag
Precondition: Middlebox does not know about client. Policy manager has a managed policy for a tag in the client's advertised policy.
Test Steps:

1. Client sends advertisement
2. Middlebox receives advertisement and searches the FTP server for a policy
3. Middlebox retrieves the managed policy

Pass/fail Criteria:

1. Middlebox applies the managed policy.

Test Number: 1-12b
Test Name: Middlebox retrieves managed policy based on MAC address
Precondition: Middlebox does not know about client. Policy manager has a managed policy for the client's MAC address.
Test Steps:

1. Client sends advertisement
2. Middlebox receives advertisement and searches the FTP server for a policy
3. Middlebox retrieves the managed policy

Pass/fail Criteria:

1. Middlebox applies the managed policy.

Test Number: 1-13
Test Name: Real-world residential network test
Precondition: Testing is done on a residential network where the middlebox has no preexisting knowledge of the IoT/normal clients being added to the network.
Test Steps:

1. Tester connects multiple IoT devices to the home network
2. Tester connects multiple non-IoT devices to the home network
3. Tester will perform normal device operations on each device for 5 minutes a piece
4. Tester will note any problems that arise that are not normal to the device.

Pass/fail Criteria:

1. Multiple clients are all sending policies without problems
2. The middlebox properly identifies the clients and applies the policies

B.2 Bad Actors Introduced

Test Number: 2-1
Test Name: Firewall does not apply policies with invalid signatures.
Precondition: Middlebox has a clean database and does not know about client.
Test Steps:

1. Client sends advertisement with a policy rule that drops 1.1.1.1 with an invalid signature.
2. Client sends DNS request to 1.1.1.1.

Pass/fail Criteria:

1. DNS request is dropped.
2. Policy is not stored within the database.

Test Number: 2-2
Test Name: Firewall does not apply policies when the new policy revision number is less than or equal to the old policy revision number.
Precondition: Middlebox has a database entry that consists of a policy with revision number 1 and a condition to drop 1.1.1.1.
Test Steps:

1. Client sends advertisement with a policy revision number of 1 and a rule that passes 1.1.1.1.
2. Client sends DNS request to 1.1.1.1.

Pass/fail Criteria:

1. DNS request is dropped.
2. Policy is not stored within the database.

B.3 Throughput Performance

Test Number: 3-1
Test Name: Throughput without our code
Precondition: Middlebox is set up for iptables forward without any filtering
Test Steps:

1. Download 4 GB of data to the client through the middlebox
2. Upload 4 GB of data from the client through the middlebox

Measurements: How long it took to transfer 4 GB

Test Number: 3-2
Test Name: Ping performance without our code
Precondition: Middlebox is set up for iptables forward without any filtering
Test Steps:

1. ping ipv4.google.com from the client
2. elapse the ping for 10 minutes
3. terminate ping

Measurements: ping averages and packet loss percentage

Test Number:3-3
Test Name: Speedtest CLI performance without our code
Precondition: Middlebox is set up for iptables forward without any filtering
Test Steps:

1. run Speedtest CLI from the client

Measurements: upload and download speed

Test Number: 3-4
Test Name: Throughput with our code
Precondition: Middlebox is running our firewall
Test Steps:

1. Download 4 GB of data to the client through the middlebox
2. Upload 4 GB of data from the client through the middlebox

Measurements: How long it took to transfer 4 GB

Test Number: 3-5
Test Name: Ping performance with our code
Precondition: Middlebox is running our firewall
Test Steps:

1. ping ipv4.google.com from the client
2. elapse the ping for 10 minutes
3. terminate ping

Measurements: ping averages and packet loss percentage

Test Number:3-6
Test Name: Speedtest CLI performance with our code
Precondition: Middlebox is running our firewall
Test Steps:

1. run Speedtest CLI from the client

Measurements: upload and download speed

B.4 Memory/CPU Performance

Test Number: 4-1
Test Name: Memory usage of the running firewall.
Precondition: Middlebox has an empty database and no clients on the network.
Test Steps:

1. The middlebox is running with no network traffic flowing through it.
2. Run the script to graph memory usage of the program.

Measurements:

- How much total memory is consumed by the process?
- How much RAM is consumed by the process?
- How much memory is shared between processes?
- What is the CPU usage?
- What percent of total RAM is available?

Test Number: 4-2
Test Name: Memory usage of the running firewall with network traffic.
Precondition: Middlebox has an empty database with one client on the network.
Test Steps:

1. The middlebox is running with no network traffic flowing through it.
2. Run the script to graph memory usage of the program.
3. Run the script to send network traffic from the client through the middlebox.

Measurements:

- How much total memory is consumed by the process?
- How much RAM is consumed by the process?
- How much memory is shared between processes?
- What is the CPU usage?
- What percent of total RAM is available?

Test Number: 4-3
Test Name: Memory usage of the running firewall with a policy applied.
Precondition: Middlebox has an empty database with one client on the network.
Test Steps:

1. The middlebox is running with no network traffic flowing through it.
2. Run the script to graph memory usage of the program.
3. Run the script to send a policy from the client to the middlebox.

Measurements:

- How much total memory is consumed by the process?
- How much RAM is consumed by the process?
- How much memory is shared between processes?
- What is the CPU usage?
- What percent of total RAM is available?