Towards a Low-Memory-Footprint, Container-Based IoT Security Gateway

Sanjay Chandrasekaran

CMU-CS-19-117
August 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Vyas Sekar, Chair
David A. Eckhardt

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Copyright © 2019 Sanjay Chandrasekaran

This research was supported by the National Science Foundation under grant number CNS-1564009. This work was also supported in part by the CONIX Research Center (SRC-JUMP 2779.009), one of the six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Keywords: IoT, Containers, Software Defined Networks, Network Function Virtualization, Docker, Snort

Abstract

Securing IoT devices is a challenge, as some devices have long deployment lives and lack an intrinsic method for updating their firmware. Vulnerabilities in IoT devices' software continue to be found, and patching each individual device's firmware is unscalable, as the number of deployed IoT devices is steadily rising. Rather than directly securing the software shipped on the device, we adopt an alternative approach by securing these devices at the network layer. Our goal is to enable an IoT Security Gateway that can provide the fine-grained, device-specific security policies that are currently missing in IoT network security, using virtualized Network Functions (vNFs). We envision (1) separate vNFs for each device to allow us to implement device-specific functionality, as well as (2) isolation between each of these vNFs. However, naively deploying separate vNFs for each device will come at the cost of additional computing resources. We analyze the memory footprint of running different vNFs and develop specific optimizations for an open-source memory-intensive vNF, Snort. We observe that significant memory goes toward large socket buffers as well as processing unnecessary rules for detecting malicious activity. We proceed by exploring both Snort-specific solutions that take advantage of Snort's open-source codebase, and generic solutions that can be applied to other types of NFs. Combining these solutions, we ultimately demonstrate the ability to increase the number of Snort instances that can simultaneously run on a low-cost gateway by at least ten-fold.

Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Vyas Sekar, for giving me the opportunity to pursue my interests, and providing me with a supportive environment to reflect and improve upon my research skills.

Next, I would like to thank Prof. David A. Eckhardt for being an amazing mentor, and being a pivotal part of my growth during my undergraduate and graduate studies. Prof. Eckhardt goes above and beyond to help his students succeed, and I am truly grateful for the lessons I've learned from him both inside and outside of the classroom.

I would also like to thank Prof. Sekar's research group for their advice and feedback along this journey. Specifically, I would like to thank:

- Matt McCormack, for his never-ending support and teamwork in tackling any problem that came my way,
- Tianlong Yu, for sparking my interest in IoT security as well as helping me find an area for contribution, and
- Grace Liu, for providing valuable inspiration and feedback during various points of this project.

I would like to thank the entire picoCTF team for all their support and patience towards my research commitments.

I would like to thank Deewane A Capella for five years of music, memories, and relaxation when things got tough.

I would like to thank all my friends at CMU for being an amazing support system, as well as all of my DotA friends for the late night fun and games.

And last but not least, I would like to thank my parents and sister for pushing me to believe in myself and helping me grow into the person I am today.

Finally, I would like to thank God for blessing me with this experience and providing me with the ability to pursue a research-driven career.

Contents

1 Introduction **1**

1.1 The Internet of Things (IoT) 1

1.2 Problems with Current Deployment 2

1.3 Potential Solutions 2

1.4 A Low-Cost Solution 4

2 Background & Related Work **5**

2.1 Network Function Virtualization 5

2.2 Software Defined Networks 5

2.3 Virtual Machines (VMs) 6

2.4 Containers 6

2.4.1 Docker 6

2.5 Snort 6

2.5.1 Community Ruleset 7

2.5.2 Deterministic Finite Automata 8

2.6 Related Work 10

2.6.1 DeadBolt 10

2.6.2 Precise Security Instrumentation (PSI) 10

2.6.3 Glasgow Network Functions (GNF) 10

3 Overview **11**
3.1 Understanding the Problem 11
3.2 Methodology 12
3.2.1 Linux Memory tools 13
3.2.2 Intel VTune 15
3.3 Design Space of Options 16
3.3.1 Shared Libraries 17
3.3.2 Device-Specific Rulesets 18
3.3.3 Socket Buffers 18
3.3.4 Sharing Common Resources 19
3.3.5 Combined Optimization 20

4 Implementation & Evaluation **21**
4.1 Specific Optimizations 21
4.1.1 Shared Libraries 22
4.1.2 Device-Specific Rulesets 23
4.1.3 Socket Buffers 25
4.1.4 Sharing Common Resources 26
4.2 Combined Optimization 27
4.2.1 Memory Footprint 27
4.2.2 Performance 28

5 Conclusions & Future Work **31**
5.1 Conclusion 31
5.2 Contributions 31
5.3 Future Work 32
5.3.1 Specialized `malloc` Implementation 32
5.3.2 Anonymously Mapped Pages 32
5.3.3 Automated Generation of Snort Rulesets 33

5.3.4 Descheduling Idle vNFs 33
5.3.5 Automated Stress Testing 33
5.4 Final Thoughts 34

Bibliography **35**

List of Figures

1.1 The similarities and differences between Related Work 3

2.1 A state machine to match on consecutive 1 bits 8
2.2 A state machine to match on consecutive 0 bits 9
2.3 A state machine to match on consecutive 1 bits as well as consecutive 0 bits 9

3.1 The amount of memory used by a few common vNFs 12
3.2 The memory breakdown of Snort based on `/proc/$pid/smaps` 15
3.3 Memory consumption analysis for Snort using Intel VTune Amplifier . . 16
3.4 The memory allocated by Snort using different-sized rulesets 17

4.1 The amount of natively shared memory between instances of Snort 22
4.2 Breakdown of Snort's community ruleset 23
4.3 The effectiveness of sharing Snort's DFA data structure 26
4.4 The amount of memory used by Snort using different optimizations . . . 28
4.5 The performance of Snort using each memory optimization 29

List of Tables

4.1 The top 10 most common tags for the Snort community ruleset 24
4.2 The number of Snort rules applicable to each of a few different IoT devices 25
4.3 The average and peak throughput for a few different IoT devices 25
4.4 The maximum number of Snort instances that can be run simultaneously . 27

Chapter 1

Introduction

In this thesis, we help enable a security gateway that uses virtualized Network Functions (vNFs) to secure IoT devices. We analyze the memory footprint and bottlenecks to running different vNFs on a low-cost hardware, and develop specific optimizations for a memory-intensive vNF, Snort (Roesch et al. [1999]).

1.1 The Internet of Things (IoT)

The world is moving towards IoT, as evidenced by the rapid industrialization and development for new smart homes and cities (Helal and Bull [2019], Bibri [2018], Silva et al. [2018]). The sheer number of network-connected devices available in these locations makes them an ideal target for security attacks, as they provide both significant computing power and bandwidth to an adversary. Attacks like Mirai (Antonakakis et al. [2017]), caused by default or unchangeable passwords, reveal a clear lack of forethought with respect to security when these devices were manufactured. Additionally, many of these devices have planned long term deployments, during which future bugs and vulnerabilities can be discovered and exploited. For example, NAS hard drives using the Samba file sharing protocol fell victim to the SambaCry attack (CVE [2017-7494]), a Linux variant of the well-known WannaCry attack (CVE [2017-0144]). Despite public disclosure of the vulnerability, prompting firmware patches released by manufacturers, unpatched devices still continued to be sold by third party vendors. Network scans using Shodan (Matherly [2015]), also expose several devices still running vulnerable Samba protocols. With the world becoming increasingly IoT dependent, we need to find a way to address the vulnerabilities within these IoT devices that surround us everyday.

1.2 Problems with Current Deployment

IoT device deployments suffer from numerous security flaws which the network is too coarse-grained to deal with (Yu et al. [2015b]). Home routers simply serve as gateways to the external network and contain one large set of firewall rules that are applied to the whole network. These firewall rules mostly address traffic between internal and external networks, and are rarely used for traffic between devices on the internal network. This broadly increases the attack surface, since an attacker would need to compromise only one device on the internal network to maneuver past the firewall and compromise more devices. It is generally difficult to rely on manufacturers' long-term support to release firmware updates for these devices, as support for the device may die off or they may be focusing their efforts on future products and streams of revenue. Additionally, patching the device immediately may not always be feasible for every end user. For example, a smart water heater may be performing critical functions for a house and its tenants, and therefore must remain online and functional until a time can be coordinated for the device to be taken offline and updated. This results in publicly documented exploits for unpatched devices, otherwise known as "one-day" exploits. These exploits can be blocked at the network level, based on a signature of the exploit. However, these rules must be manually added to each individual router by a knowledgeable end user, which can be difficult to automate and does not realistically scale. Due to these issues, we lack a reliable method to deal with vulnerabilities in IoT devices, even after they become public knowledge.

1.3 Potential Solutions

There have been a number of approaches taken to try and fix the issues with IoT device deployments. A common theme among some of these solutions is an IoT-specific gateway that uses virtualized Network Functions (vNFs) (Guerzoni et al. [2012]) to provide extra functionality for devices on the network.

- Ko and Mickens [2018] propose DeadBolt, a specialized access point that requires IoT devices to remotely attest that they are running up-to-date and trusted software; the AP uses virtual device drivers, similar to vNFs, for IoT devices that are unable to attest.

- Yu et al. [2017] present Precise Security Instrumentation (PSI), which aims to provide the network with more context through the ability to customize dynamic policies for different devices. These policies can trigger events such as spawning a

Intrusion Prevention System (IPS) as a vNF.

- Cziva [2018] proposes Glasgow Network Functions (GNF), a container-based NFV framework for dynamically allocating vNFs at the edge of the network, and can be used for IoT devices and more.

The similarities between these works are summarized in Figure 1.1. Both Yu et al. [2017] and Cziva [2018] enforce isolation between the vNFs for devices on the network. This is mainly to ensure that there is no interference among policies for different devices that could potentially leave the network vulnerable. We aim to provide isolation between IoT devices, and run a separate instance of each NF, for every device connected to the gateway.

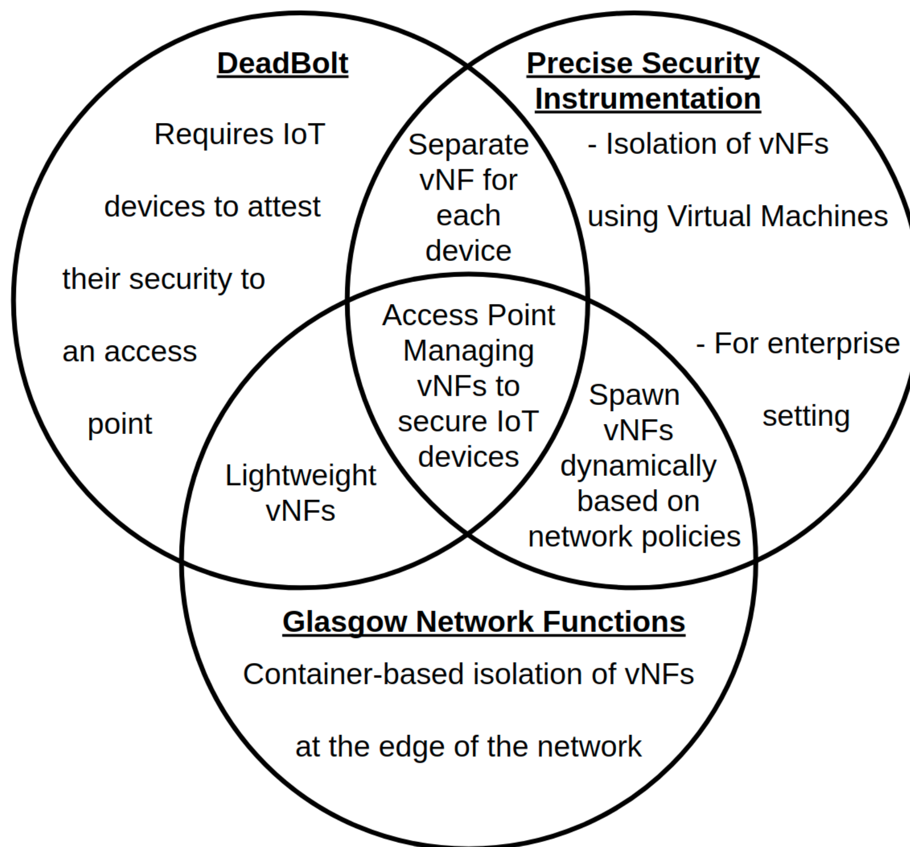

Figure 1.1: This figure shows the similarities and difference between network security solutions that apply to IoT devices.

1.4 A Low-Cost Solution

Datacenters and enterprises have already attempted to use a vNF-per-device model, however these settings are not usually resource constrained. Given the increasing number of IoT attacks targeting home users, we want this model to be feasible for use in the common household as well, with respect to both cost and usability. We envision an IoT Security Gateway that provides the fine-grained, device-specific security policies that are currently missing in IoT network security. We incorporate isolation as a design constraint, by spawning a separate instance of each vNF, for each device, rather than a single instance of each vNF, for the entire network. We run each vNF inside a container, as containers are a lightweight solution for isolating multiple vNFs on a single device compared to Virtual Machines (VMs). We believe that the cost of this gateway should be at a comparable price to the IoT devices being protected to promote adoption.

Unfortunately, we find that we are limited on the number of vNFs we can run on a low-cost gateway, due to the amount of memory (RAM) on the device. In this thesis, we analyze the memory footprint of running different vNFs on a low-cost gateway, and determine memory bottlenecks to scaling to more IoT devices. We develop specific optimizations for an Intrusion Detection System/Intrusion Prevention System (IDS/IPS), Snort (Roesch et al. [1999]), and are able to increase the number of Snort instances that can simultaneously run on a low-cost gateway by at least ten-fold, with no reduction to performance. In Chapter 2 we provide some background on the components of our security gateway along with the related work from which we drew our motivation. In Chapter 3 we analyze the memory footprint of multiple vNFs and develop optimizations for a specific memory-intensive vNF, Snort. In Chapter 4 we evaluate our optimizations to Snort with respect to memory and performance. Finally, in Chapter 5 we conclude and present future research directions that could further reduce the memory footprint of Snort.

Chapter 2

Background & Related Work

2.1 Network Function Virtualization

Advances in Network Function Virtualization (NFV) allow us to use virtualized network functions (vNFs) where we would normally deploy separate middleboxes. There are a number of advantages to using vNFs over middleboxes, explained in Yu et al. [2015a]. One important benefit of vNFs is that software can be flexibly reconfigured at a trivial cost to the user when compared to middleboxes, which normally require updates from their manufacturers to change their behavior. This is very useful for IoT devices, which tend to have varying needs. For example, an IoT device that is shipped with a hard-coded password can be reconfigured at the network level so that it can be accessed only through a securely authenticated proxy. vNFs provide necessary functionality to enable a secure and efficient network.

2.2 Software Defined Networks

Routing traffic for a device through its respective vNFs can be simplified using software defined networks (SDNs). SDNs (McKeown [2009]) are growing technologies that separate the data plane and the control plane of the network. In an SDN there is a central controller that takes care of all the control and routing logic, and thus the software defines the network. Cloud services allow us to run an SDN controller remotely, reducing the resource constraints on the network while still providing flexible and dynamic route management.

2.3 Virtual Machines (VMs)

VMs (Popek and Goldberg [1974]) are one way of emulating an OS and provide isolation from other VMs as well as the host they are running on. VMs can be very useful for simulating an architecture that differs from the host's architecture. Since VMs virtualize the hardware necessary to emulate an OS, they are a resource-heavy solution for running a single isolated application per VM.

2.4 Containers

Containers (Lezcano et al. [2008]) are a lightweight solution to providing isolation between applications. Containers provide the abstraction of a separate OS from the host, using namespaces and sharing libraries with the kernel. Unlike VMs, containers do not reimplement hardware functionality, and simply isolate groups of processes running on the host OS. This provides us with a lightweight and efficient way of running isolated Linux processes on a Linux host.

2.4.1 Docker

Docker (Docker Inc. [2013]) is a well-documented and widely-available container solution. Using a *Dockerfile*, similar to a *Makefile*, for the build process, we are able to create Docker images containing newly built libraries and applications. We use Docker as a benchmark environment, as it provides us with a convenient way to build, deploy, and interact with containers. However, Docker may not be right choice, with respect to security, for our IoT gateway. There has been research that discusses potential security risks associated with using Docker containers to achieve secure isolation (Bui [2015], Combe et al. [2016]). Arnautov et al. [2016] proposes a method for secure isolation of Linux containers using Intel Software Guard Extensions (SGX) (Intel Corporation [2013], Costan and Devadas [2016]), and a similar method will likely need to be implemented for our security gateway.

2.5 Snort

Snort (Roesch et al. [1999]) is an open source Intrusion Detection System/Intrusion Prevention System (IDS/IPS) with a number of packet analysis features to generate alerts

or drop packets. Snort can be used to detect a variety of network attacks and probes in real-time based on a signature or regular expression match. For example, consider a Linux network attached storage device (NAS) that is publicly accessible to the internet and serves files to its users. A exploit for the NAS, such as the SambaCry attack (CVE [2017-7494]), may be publicly released, leaving the device vulnerable to attackers. Snort can be used to match on the signature of the exploit and drop the relevant packets. This can provide a temporary patch for the device until a patch release by the manufacturer, limiting the amount of time that the IoT device is vulnerable to the public, while still allowing users to access their files.

2.5.1 Community Ruleset

Snort offers a community ruleset that is comprised of rules that have been submitted by members of the open-source community as well as vendors that bundle and distribute Snort or Snort rules in their products. Snort maintains a community ruleset that is available to the public and updated daily. Snort also offers a ruleset for registered users which contains a snapshot of rules for a specific version of Snort. Finally, for commercial applications, Snort offers a subscriber ruleset containing rules developed, tested, and approved by the Talos Security Intelligence and Research Team (Talos), as well as the community ruleset that is updated daily.

Snort rules are classified under one of five states, to indicate their relevancy under different network policies:

- Connectivity over Security - designed to favor device performance over security
- Balanced - balance of security needs and performance
- Security over Connectivity - designed for protected networks with low bandwidth and high security requirements
- Maximum Detection - not optimized for performance, may raise false positives
- No Policy - case by case basis, added based on product name, type, or relevant CVEs

In the rest of this paper, we use the Snort community ruleset for registered users (snapshot-29120) (The Snort Project [2019a]) for our benchmark analysis, as well as to demonstrate our memory optimizations to Snort.

2.5.2 Deterministic Finite Automata

We can use a state machine to represent a single Snort rule that contains a regular-expression matching, explained in Ficara et al. [2008]. A simplified example, using an alphabet of $\{0,1\}$ and matching on consecutive 1 bits, is shown in Figure 2.1. For any input stream, we start at state a, the initial state, and process a set of bits. For example, if we see the bit sequence `<1,0,1>`, then we will start at state a, and follow the state transitions

```
a -> b -> a -> b
```

Therefore, we end up at state b and will not generate an alert.

However, if we see a bit sequence with consecutive 1 bits, such as `<0,1,1>`, then we follow the state transitions

```
a -> a -> b -> c
```

We end up at state c and an alert will be generated.

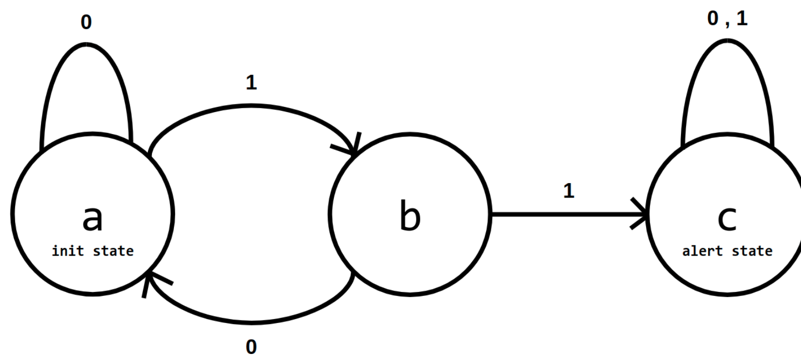

Figure 2.1: An example of a simple state machine with an alphabet of $\{0,1\}$. This state machine represent a single rule, and will raise an alert when given an input containing consecutive 1 bits.

Similarly, Figure 2.2 shows a complementary state machine which raises alerts on inputs with consecutive 0 bits. These rules can be combined into a single state machine, shown in Figure 2.3, that raises alerts for consecutive 1 bits and for consecutive 0 bits.

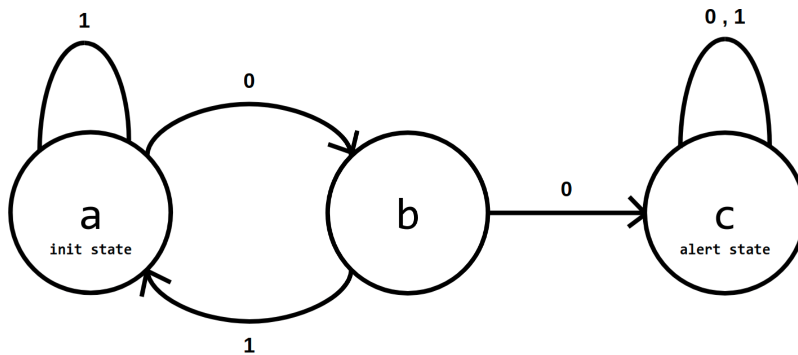

Figure 2.2: An example of a simple state machine with an alphabet of {0,1}. This state machine represents a single rule, and will raise an alert when given an input containing consecutive 0 bits.

These three state machines are Deterministic Finite Automata (DFA), because at each state, there exists a transition to a new state, for every input symbol in the alphabet, which is only {0,1} in these simplified examples. Snort combines state machines for multiple rules to create DFAs for different protocols, each with an alphabet of byte values {0-255}, and can process packets in *O(n)* time, where *n* is the number of bytes in the packet. For each packet, we start at the initial state in a state machine and transition to each subsequent state, raising an alert or generating a drop in *O(n)* time, based on the bytes in the packet. This is noteworthy because the per-packet latency is unaffected by the size of the Snort ruleset, which can grow to be very large. Snort uses the Aho-Corasick algorithm, described in Norton [2004], to combine multiple state machines into a single DFA.

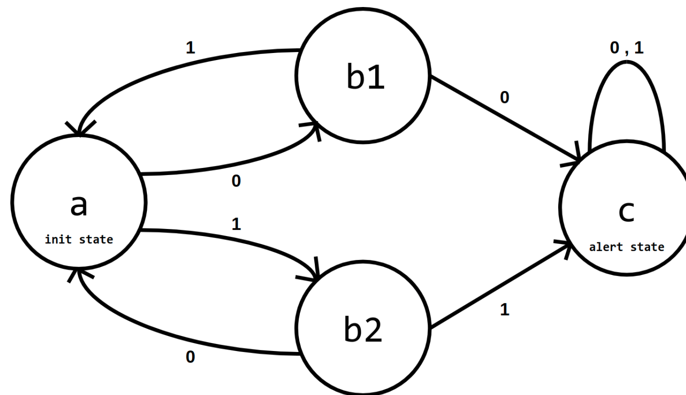

Figure 2.3: An example of a simple state machine with an alphabet of {0,1}. This state machine will match on consecutive 1 bits as well as consecutive 0 bits.

2.6 Related Work

2.6.1 DeadBolt

DeadBolt is a security framework, by Ko and Mickens [2018], for managing IoT devices. The system uses a specialized access point that requires IoT devices to remotely attest that they are running up-to-date and trusted software. The DeadBolt AP uses virtual device drivers for IoT devices that are unable to attest. These virtual device drivers are similar to vNFs, and provide functionality such as intrusion detection or encryption, that is needed by IoT devices.

2.6.2 Precise Security Instrumentation (PSI)

PSI is an enterprise security architecture, by Yu et al. [2017], that aims to provide the network with more context through the ability to customize dynamic policies for different devices. PSI aims to provide the network with more precision with respect to (1) *isolation* to ensure we do not have interference among network policies; (2) *context* to customize policies for different devices; and (3) *agility* to dynamically act in response to network events. Combined with a detection mechanism, PSI can provide us with the ability to dynamically spawn vNFs within the network, to provide necessary functionality, needed to secure the network.

2.6.3 Glasgow Network Functions (GNF)

GNF is a comprehensive framework, proposed by Cziva [2018], that manages lightweight, container-based vNFs for devices on the network. GNF provides a latency-optimal vNF-placement solution that can be used dynamically over the network. This system manages a number of simultaneously running vNFs and demonstrates multiple use cases, such as IoT DDoS mitigation and on-demand trouble shooting for providers.

Chapter 3

Overview

Our goal is to lower the memory footprint of vNFs so that they can be run on low-cost hardware. We use a Raspberry Pi 3B+ (Raspberry Pi [2018]) with 1 GB RAM and 1 GB swap space as the hardware for our security gateway, as it is both a low-cost and widely available solution. In Section 3.1 we explain issues we encounter when running vNFs in a per-device configuration on low-cost hardware. In Section 3.2 we explain generic methods for measuring the memory footprint of a vNF using open-source tools. In Section 3.3 we explain our approach to creating a more scalable version of a specific memory-intensive vNF, Snort. In Chapter 4 we present our results on how much memory is saved using each of these optimizations.

3.1 Understanding the Problem

The average US household currently has around eight IoT devices according to Cisco [2019], and we can expect this number to grow in the future. To enable our security gateway, we must be able to support a vNF-per-device model, for each of these IoT devices. Without any optimizations, we can run only three instances of our containerized version of Snort on a Raspberry Pi, which is not enough to support one vNF per device for the average US home's IoT deployments. The main issue we encounter when spawning multiple Snort instances is an inadequate amount of memory (RAM) on the device. We measured the memory footprints for a single instance of four commonly used vNFs: `snort`, `squid`, `iptables`, and `openvpn` using the Linux `free` tool (further described in Section 3.2.1). We also measure the memory footprints, for a single instance of each of these vNFs, within a Docker container, to achieve isolation between vNFs. The memory foot-

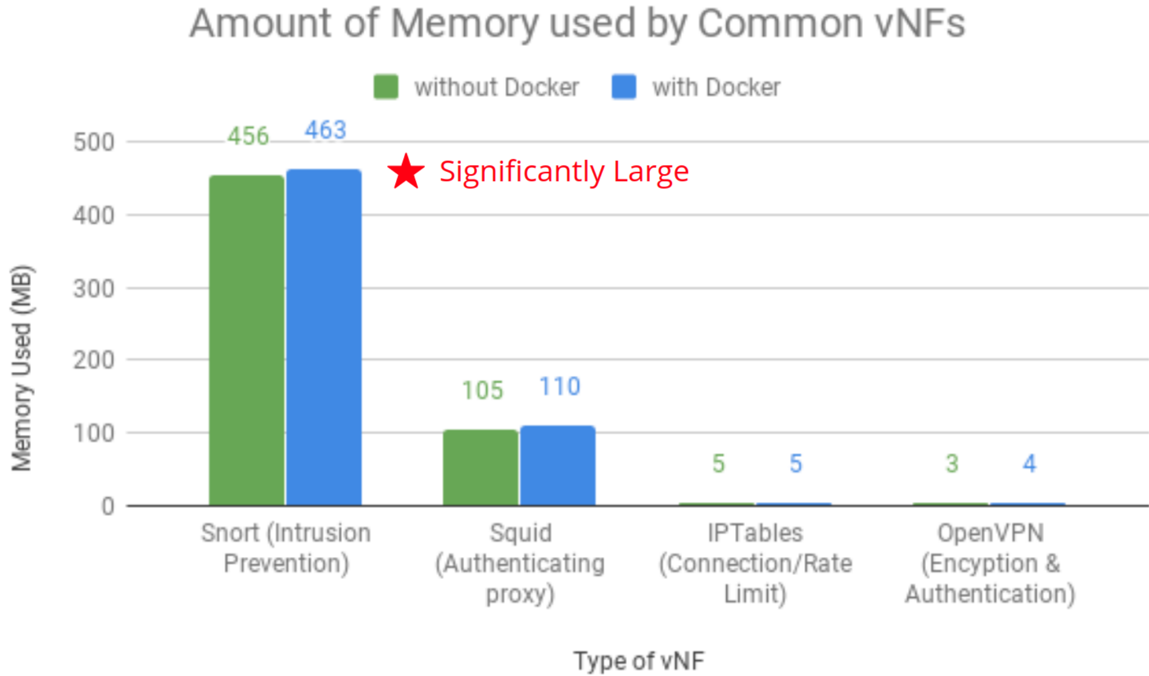

Figure 3.1: The amount of memory used by a single instance of a few common vNFs, each running inside a Docker container. These memory footprints were measured using the Linux `free` utility. We notice that the memory footprint of Snort is significantly larger than the other applications given a Raspberry Pi 3B+ only has 1 GB RAM.

print for each of the vNFs is shown in Figure 3.1. We observe that Snort uses a significant amount of memory compared to other vNFs. Snort, described in Section 2.5, is an Intrusion Detection System/Intrusion Prevention System (IDS/IPS) with a variety of packet analysis features to generate alerts or drop packets based on a signature. Specific rules can be added to Snort to drop packets that are part of an exploit, based on a signature, to secure an unpatched IoT device. Therefore we focus our efforts on reducing the memory footprint of Snort, to allow us to run an isolated instance of Snort for each IoT device on the network, on our security gateway.

3.2 Methodology

In this section, we analyze the memory footprint of Snort (v2.9.12), using the Snort community ruleset for registered users (snapshot-29120). We ran Snort using the `afpacket`

packet acquisition model with the default DAQ buffer size of 128 MB. We ran each Snort instance within a Docker container (v18.09.0), and connected the container to two Docker networks for inbound and outbound traffic. We used multiple tools, described later in this section, to measure the memory footprint of spawning Snort instances within Docker containers. These methods can be generalized to determine the memory bottleneck for scaling any vNF.

3.2.1 Linux Memory tools

We use existing Linux tools to profile the memory footprint of Snort. We mainly use the Linux `free` tool which parses information from the file `/proc/meminfo`. We also parse the information in `/proc/$pid/smaps`, where `$pid` is the process identity number of the running application. Finally, we use Intel VTune Amplifier (Intel [2017]) to examine the source code of the application and assess the room for optimization.

The Linux `free` tool

Linux's `free` tool is the simplest tool for measuring the memory footprint of an application. It provides a summary of the information in `/proc/meminfo` for RAM as described in Section 1 of the `man` page for `free` (Linux [2018]):

`total`	Total installed memory
`used`	Used memory
`free`	Unused memory
`shared`	Memory used (mostly) by tmpfs
`buffers`	Memory used by kernel buffers
`cache`	Memory used by the page cache and slabs
`available`	Estimation of memory available for new applications

Linux's `free` tool also provides us with the `total`, `used`, and `free` values for swap memory as well. We can verify `total` as:

$$\texttt{total_memory} = \texttt{used} + \texttt{free} + (\texttt{buffers} + \texttt{cache})$$

We calculate the memory footprint as follows:

$$\texttt{memory_footprint} = \texttt{used} + \texttt{swap_used}$$

We notice that `shared` is not accounted for in `total_memory` and is distributed between `used`, `swap_used`, `buffers` and `cache`. Therefore for applications that allocate shared memory we calculate the memory footprint as follows:

$$\texttt{memory_footprint}_{\texttt{shared}} = \texttt{used} + \texttt{swap_used} + \texttt{shared}$$

It is not appropriate to add the entire `buffers` and `cache` to the footprint because it may not all be associated with the application being measured.

/proc/meminfo

Linux provides a memory breakdown for the system in `/proc/meminfo`. This output provides more detailed information about the `used` and `free` memory, based on the kernel's paging system and memory allocations. The memory information provided is based on the number of physical pages allocated by the system. This means that a C program that allocates an 8 GB buffer, but only writes to 1 byte of that buffer, will only indicate a memory footprint of 1 physical page size. This is because most OSes often implement an allocate-on-write optimization, where pages will not be allocated to a process until they are actually used for a read or write action. The relevant fields to measure the memory footprint of an application using `/proc/meminfo` are already parsed and summarized by Linux's `free` tool.

/proc/$pid/smaps

Linux allows us to get the virtual memory breakdown for a single process, as well as which segments of memory are shared with other processes. This output is especially useful because it further indicates the library or process that allocated each segment in memory. For example, buffers allocated for a socket connection, or virtual pages allocated for a shared library, are indicated in the output. This breakdown also indicates which pages are shared and which are private to a process.

When measuring Snort, we see memory segments allocated for Snort code, shared libraries, stack space, heap space, and socket buffers. There are also memory segments allocated anonymously that we have not attributed to a single part of an application. Our results for the memory breakdown of Snort are shown in Figure 3.2.

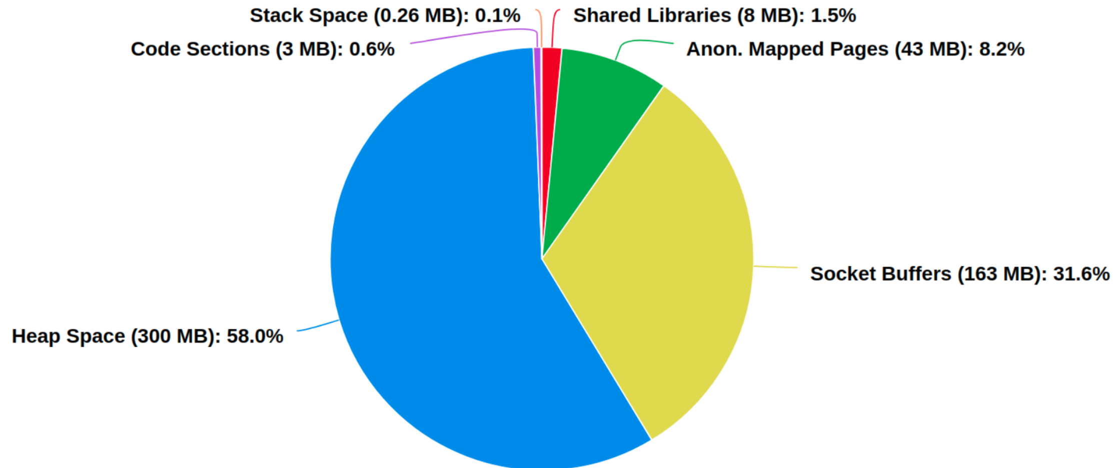

Figure 3.2: This figure shows the memory breakdown for the components of Snort based on the information in `/proc/$pid/smaps`. The anonymously mapped pages are allocated from calls to `mmap`, using the `MAP_ANONYMOUS` flag and have not been attributed to any single component of Snort.

3.2.2 Intel VTune

Intel VTune Amplifier is a great tool that analyzes performance and memory bottlenecks for programs compiled with debug information. This approach requires source code for the application being analyzed. Fortunately, we have access to this information for Snort because it is open source. We analyze the memory consumption of Snort using this tool and the results are shown in Figure 3.3. This tool allows us to trace back through each memory allocation to determine the context of its usage. This allows us to design application-specific solutions to lower the memory footprint of each instance, and make the application more scalable.

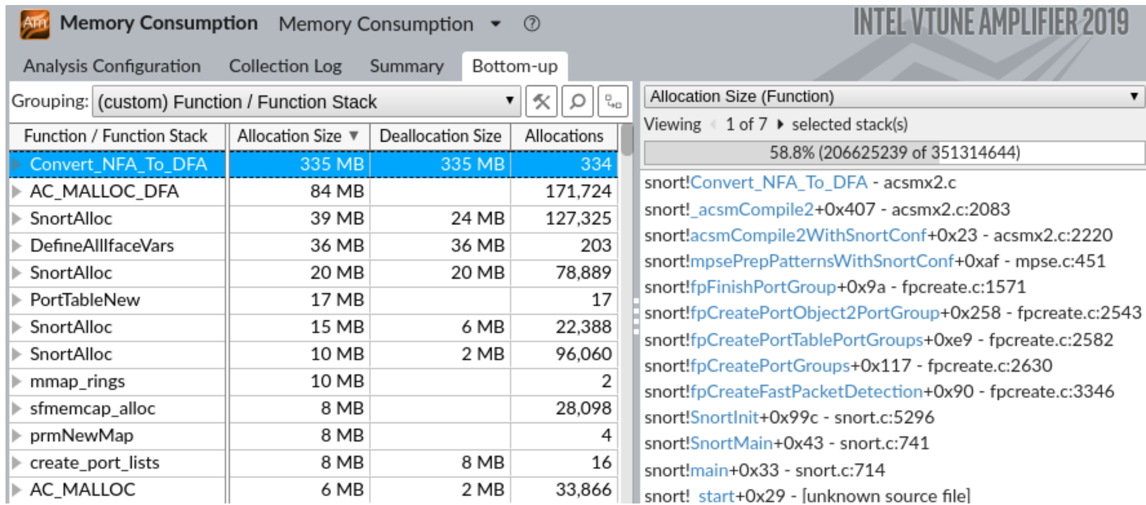

Figure 3.3: This figure shows the results of running Intel VTune Amplifier's memory consumption analysis on Snort. We can see the amount of memory allocated by each function in the code. In the right-most column is the function traceback of the `Convert_NFA_To_DFA` function (selected), which allocates a majority of the memory (335 MB) in the default configuration of Snort.

3.3 Design Space of Options

Our goal is to investigate solutions to reduce the memory footprint of Snort. Specifically, we try to look for solutions that could be easily applied to other vNFs used by our security gateway. We looked at 4 specific areas for optimization:

- Sharing application-level shared libraries and binaries across containers (Section 3.3.1)
- Using device-specific rulesets (Section 3.3.2)
- Reducing the size of socket buffers (Section 3.3.3)
- Sharing common resources across containers (Section 3.3.4)

In this section, we describe our approach for each of these optimizations, and evaluate the memory footprint of Snort using each of these optimizations in Section 4.1.

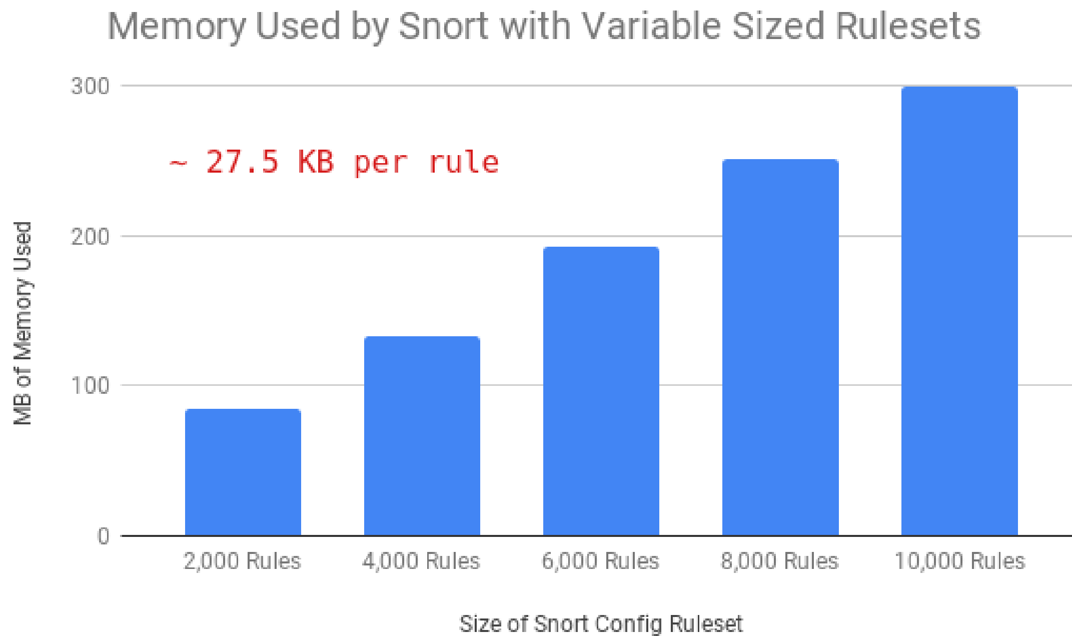

Figure 3.4: This figure shows the amount of memory allocated by Snort using different-sized rulesets, generated as a subset of Snort's community ruleset. The amount of memory used appears to be linearly correlated to the size of the ruleset, therefore we can estimate the memory per rule to be 27.5 KB.

3.3.1 Shared Libraries

Initially, we hypothesized that shared libraries may contribute heavily to the memory footprint, and could be a potential area for optimization. The same shared libraries would be built and loaded into memory redundantly across separate Docker containers, because the OS is unaware that these libraries are identical. We hypothesized that mounting all shared libraries and binaries onto a shared volume would be one way to signal to the OS that they are the same object, and prevent the shared library from being loaded into memory multiple times. We evaluate the amount of memory saved using this optimization in Section 4.1.1.

3.3.2 Device-Specific Rulesets

Next, we looked at the rulesets deployed with Snort. We see that a significant amount of memory is allocated on the heap according to Figure 3.2. We measured the amount of memory allocated on the heap for Snort configurations with different sized rulesets, shown in Figure 3.4. We find that the memory allocated on the heap is correlated to the size of the ruleset that Snort is deployed with. We are running Snort with the community ruleset for registered Snort users (The Snort Project [2017]), which is originally meant to deployed for one instance of Snort at the edge of a network. This ruleset is meant for devices of all types; therefore, it contains a laundry list of rules. However, all of these rules cannot be applicable to a single device, since the relevance of a rule varies based on OS, device type, and even manufacturer of that device. Therefore, we can build and deploy our per-device Snort instances with rules that are applicable only to the device it is protecting, reducing the size of each ruleset, thus lowering the total memory footprint of each Snort instance. To facilitate deployment, we need a way to query relevant rules based on a set of criteria. We use a `sqlite3` database to keep track of each attribute for all the community rules. We can automatically tag these rules based on keywords, as well as scrape websites for information about what the rule is matching on. Using these tags, we can create trimmed rulesets based on the type and functionality of an IoT device. We evaluate the utility of our tagging system in Section 4.1.2.

3.3.3 Socket Buffers

The other main contribution to the memory footprint of Snort comes from memory allocated for socket buffers, as shown in Figure 3.2. The default buffer size is 128 MB, however we observe that 163 MB of memory is allocated for socket buffers. This is due to the overhead for the "afpacket header" (66 bytes) that gets added to the allocated buffer size, explained in Section 1.5.3 of the Snort manual (The Snort Project [2019b]). This accounts for the 163 MB allocated for socket buffers, shown in Figure 3.2, even though the default buffer size is only 128 MB. We can calculate the size of the memory allocated for a buffer of size s, given a frame size of e bytes, as follows:

$$\texttt{new_frame_size} = e + 66$$

$$\texttt{num_frames} = \frac{s}{\texttt{new_frame_size}} = \frac{s}{e + 66}$$

Memory is allocated for a process in blocks, where each block has a size that is a multiple of the system's page size. The smallest block size that can fit at least one frame is 4 KB

(4096 bytes), which can fit 2 frames (with size = `new_frame_size`) per block.

$$\texttt{mem_allocated}_{\text{KB}} = \texttt{num_frames} * \frac{\texttt{block}}{\texttt{2 frames}} * \frac{\texttt{4KB}}{\texttt{block}}$$

$$\texttt{mem_allocated}_{\text{KB}} = 2 * \frac{s}{e + 66}$$

Therefore with a frame size of 1518 bytes, a 128 MB buffer would have an overhead of $\sim$ 35 MB.

The default buffer size of 128 MB for Snort is significantly larger than one second of peak bandwidth for any IoT device. The socket buffers allocated by Snort are normally meant to buffer all the packets entering the network, and are unnecessarily large for a single IoT device. For example, the peak bandwidth measured for a Smart Light Bulb is 24 Kbps and occurs for only milliseconds at a time. We will allocate a separate buffer for each Snort instance, and determine the appropriate size depending on the respective IoT device's traffic patterns. In Section 4.1.3, we show the average and peak throughput for a few IoT devices to determine a reasonable size for a socket buffer that only holds traffic for a single IoT device.

3.3.4 Sharing Common Resources

We looked into the source code of Snort based on the results in Figure 3.3, and we find that a large amount of memory is consumed to create Deterministic Finite Automata (DFA). Snort uses a DFA to process packets, as explained in Section 2.5.2. This results in a packet processing time that is proportional to the length of the packet. Unfortunately, this comes at a tradeoff because a large amount of memory is used to store a lookup table containing state transitions for the DFA. This DFA is used in a read-only manner, however it is not inherently shareable across Snort instances. The goal of this solution is to create a single DFA in read-only memory for all Snort instances to access and use through inter-process communication (IPC). Snort creates its DFA using the list of rules provided in the Snort configuration. Most of these rules match only on the payload of the packet, and have no constraints on the source or destination IP address. If we naively take the union of all of the devices' Snort configurations to create a single shared DFA, then any rule without source or destination IP address constraints will match on traffic for all the devices on the network. This would result in a Snort instance raising an alert or dropping a packet for a rule that is irrelevant to the IoT device it is processing packets for. Since all the IoT devices will be using our gateway as an access point, we will know the IP address of each IoT device on the network. Therefore, we can reflect this in the union of the

Snort configuration files, where each rule will match on the IP addresses of the devices for which that rule is applicable to. While this will create a slightly more complex DFA, we hypothesize that this method of sharing a single DFA consumes less memory as we run more instances, as opposed to creating a separate DFA for each Snort instance, some of which may be matching on a common subset of rules.

Snort is implemented in C, which means there is no native deepcopy implementation for the DFA data structure, which is comprised of a number of pointers. We found that the most efficient way to get this data structure into shared memory is to allocate it into shared memory when it is initially created, rather than deepcopy the data structure after it is allocated. We can change the relevant `malloc` calls in the Snort code to `shared_malloc` calls, that allocate the DFA on a shared heap. We build our `shared_malloc` on top of an existing glibc `malloc` implementation (Lea [2011]). We allocate an area in shared memory that we can use as a new heap to be shared between processes. In the implementation of `malloc`, we replace calls to the `sbrk` function, which allocates new memory to a process, with a new function that simply returns a pointer to an area in the shared memory we initially allocated. Each Snort instance will then map this shared memory into its own virtual address space and use the DFA data structure in a read-only manner. The code for our implementation of Snort using `shared_malloc` calls to allocate a DFA on a shared heap can be found at

`https://github.com/speeeday/shared-dfa-snort`

This method can also be generalized to any vNF with costly read-only data that does not need to be duplicated for every instance of the vNF. We evaluate the memory saved using this implementation of Snort in Section 4.1.4.

3.3.5 Combined Optimization

Each of these optimizations attempts to reduce the amount of memory used by a certain component of Snort. We look to optimize the memory footprint of the shared libraries, socket buffers, as well as memory allocated on the heap. These optimizations can also be combined to achieve a further-reduced memory footprint. We evaluate each of these optimizations as well as their combined memory footprint in Sections 4.1 and 4.2 respectively.

Chapter 4

Implementation & Evaluation

The following are our findings from investigating potential solutions to lower the memory footprint of Snort. We hope that these optimizations allow us to support a separate Snort instance for at least the average number of IoT devices per U.S. household. In addition to memory, we test our optimizations with respect to performance to ensure we have not induced extra latency in Snort's packet processing. All of these experiments were run on a Raspberry Pi 3B+ with 1 GB RAM and 1 GB swap space and each instance of Snort running within a Docker container. The memory measurements were taken using Linux's `free` tool, described in Section 3.2.1.

4.1 Specific Optimizations

In this section we evaluate the memory footprint of Snort using each of the four optimizations explained in Section 3.3:

- Sharing application-level shared libraries and binaries across containers (Section 4.1.1)
- Using device-specific rulesets (Section 4.1.2)
- Reducing the size of socket buffers (Section 4.1.3)
- Sharing common resources across containers (Section 4.1.4)

We implemented each of these optimizations over Snort (v2.9.12) and the Snort community ruleset for registered users (snapshot-29120). We ran Snort using the `afpacket`

packet acquisition model and compare our optimizations to Snort's default DAQ buffer size of 128 MB. We ran each Snort instance within a Docker container (v18.09.0), and connected the container to two Docker networks for inbound and outbound traffic.

4.1.1 Shared Libraries

In this section we follow our intuition from Section 3.3.1 and evaluate amount of memory shared between application-level shared libraries and binaries across Docker containers. We measured the amount of shared memory between multiple instances of the same Docker image (labeled as *same image*), multiple instances of different Docker images that use identical libraries and binaries built in separate containers (labeled as *diff image*), and multiple instances of a Docker image that use the same libraries and binaries mounted on a single shared volume (labeled as *shared volume*).

The results for these experiments are shown in Figure 4.1. In two different images of the same Snort application, *diff image (snort)*, we see that none of the pages are shared for the Snort application. We also notice that in all three cases, the most shared memory is when there is one instance running. We suspect this is due to an OS-level optimization

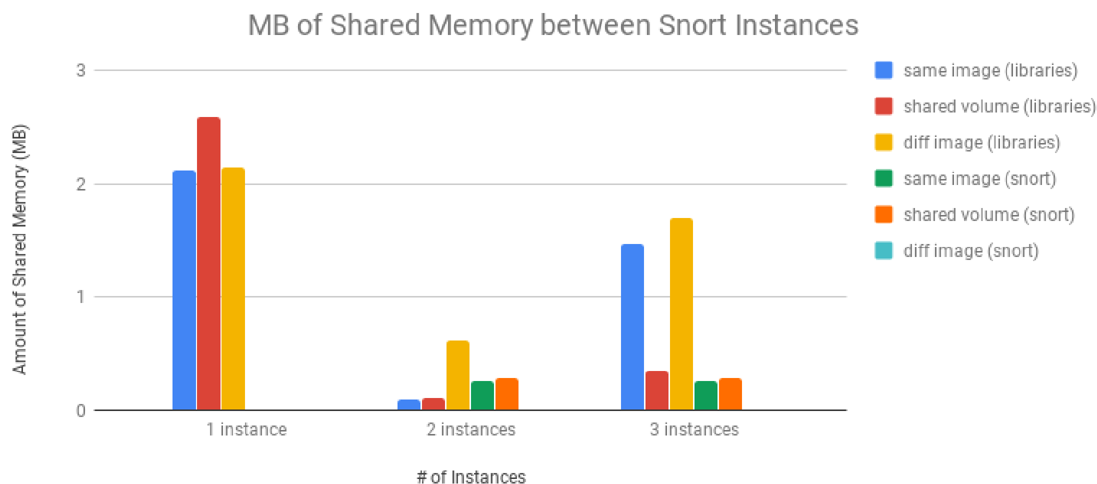

Figure 4.1: This figure shows the amount of shared memory used by shared libraries and Snort code for n instances in three cases: 1) The same Docker image of default Snort, 2) Different Docker images containing the same default Snort, 3) Docker image with all Snort files/libraries placed in a shared volume.

that causes the kernel to mark an area as shared memory until it recognizes a need for distinct copies of those physical pages, after which it proceeds to duplicate them. According to Figure 3.2, the shared libraries make up a less than 2% of Snort's memory footprint. Therefore, by Amdahl's Law, there is little potential gain to optimizing such a negligible portion of the footprint, and we focus our efforts towards more memory consuming portions of the application.

4.1.2 Device-Specific Rulesets

In this section we evaluate the effectiveness of creating device-specific rulesets from the rule-tagging system described in Section 3.3.2. We can see the breakdown of the community ruleset distributed by Snort, in Figure 4.2. We further classify these rules based on OS, device type, and manufacturer by scraping any relevant reference URLs provided by rule authors. For example, VirusTotal (VirusTotal [2012]) provides us with the sandbox behavior of different malware samples which we can use to determine which OS the malware is targeting. Additionally, websites such as `helpx.adobe.com/security`

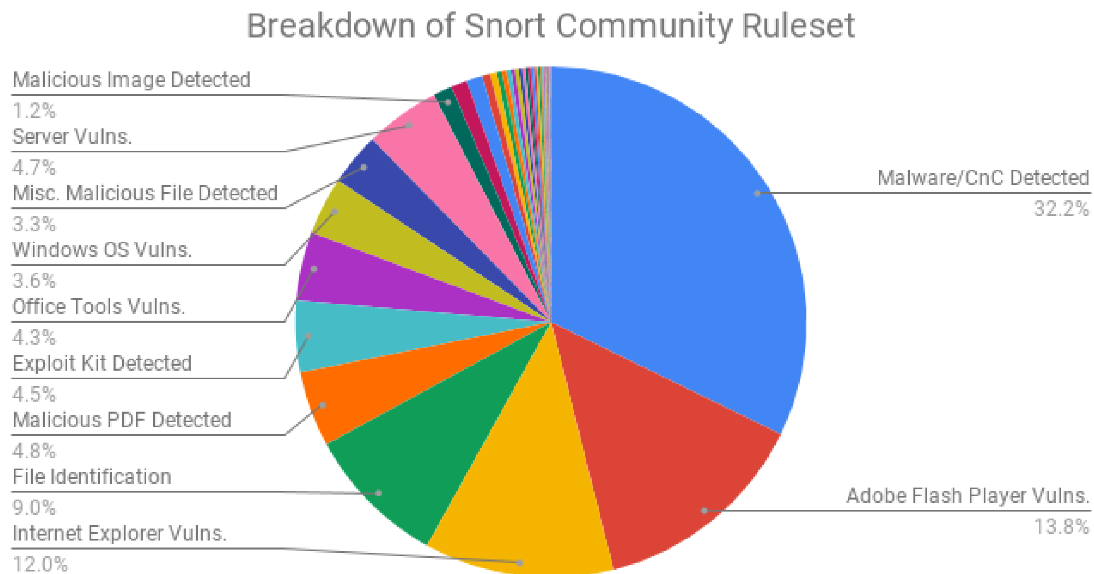

Figure 4.2: This image shows the breakdown of Snort's community ruleset which contains 10,918 rules.

contain explanations and applicability of security vulnerabilities found within their software. We use 140 tags to classify each of the 10,918 Snort community rules. The tagged ruleset and code for tagging is available at

`https://github.com/speeeday/custom-snort-rulesets`

The top ten most common tags are shown in Table 4.1.2.

Table 4.1: This table shows the top 10 most common tags from our tagging scheme for the Snort community ruleset for registered users, which contains 10,918 rules.

Tag Name	Description	# Applicable Rules
windows	for Windows-based devices	5624
malicious indicator	indicates malicious activity	3699
outbound connections	detects suspicious outbound connections	2102
adobe flash	for devices that use Adobe Flash	1582
malicious file	detects potentially malicious file attachments	983
adobe acrobat reader	for devices that use Adobe Acrobat Reader	539
osx	for macOS-based devices	517
office tools	for devices that use Microsoft Office tools	511
file attachment detected	log file attachment type	1144
file magic detected	detect and log file magic numbers	250

We utilize network scanning to automatically determine the applicability of each tag. Using `nmap` (Lyon [2009]), a network scanning tool, we attempt to identify the OS, port services, and manufacturer of each IoT device. When `nmap` is unable to provide us with this information, we attempt to look up the device vendor based on the existing MAC address using a community maintained database such as Stiller [2019], which is able to correctly identify the vendor of IoT devices in our lab. We then use this information to determine which tags are inapplicable to each IoT device on the network. We can then generate a custom ruleset using a SQL query that negates each of the inapplicable tags, removing those rules from the ruleset. The size of the rulesets for a few types of IoT devices, generated using our tagging scheme, are shown in Table 4.1.2. A bulk of the rules are meant for Windows, an OS that is not commonly used for IoT devices. Therefore, for most IoT devices, which run a Linux-based OS, we can achieve a five-fold reduction on the number of rules deployed with Snort.

Table 4.2: This table shows the number of rules present in the Snort configuration for a few different type of IoT devices. Each of these configurations is a subset of Snort's default community ruleset containing 10,918 rules.

IoT Device Type	# Relevant Rules	% of Community Rules
Linux NAS	2199	20.1
D-Link Camera	1697	15.5
Windows IoT Controller	7173	65.7
Android Smart Device	1614	14.8

4.1.3 Socket Buffers

In this section, we follow our intuition from Section 3.3.3 to reduce the size of Snort's socket buffer to a reasonable size for IoT devices. We estimate a reasonable size to use for IoT devices based on measuring the packet throughput and peak packet rates for a few IoT devices in our lab, shown in Figure 4.1.3. The minimal size for the packet buffer allowed by Snort is 1 MB which will suffice for most IoT devices. This buffer is easily configurable at runtime, and can be altered for devices with higher average throughput.

Table 4.3: This table shows the average and peak throughput for a few IoT Devices, measured in our lab using Wireshark (The Wireshark Team [2019]), from packet captures that were generated from normal use of each IoT device.

IoT Device	Avg Throughput (bps)	Peak Throughput (bps)
Nest Thermostat	759	368K
D-Link Camera	182K	1.3M
Amazon Alexa	669K	17.5M
Hue Light	6.3K	24K
Smart Plug	8.2K	78K

4.1.4 Sharing Common Resources

In the section we evaluate our approach described in Section 3.3.4 of sharing a single DFA across all Snort instances on a device. The results of sharing a DFA across all Snort instances is shown in Figure 4.3. We are able to determine the overhead per shared Snort container to be 188 MB. Each container is run with the default 128 MB buffer, as well as mutable data structures that are derived from the DFA. Additionally, the Shared DFA requires close to 80 MB more of memory to create, than the DFA for the default Snort. This is due to the extra source IP address constraints added to the Snort configuration, explained in Section 3.3.4, as well as the difference in implementation between the default `malloc` calls and the `shared_malloc` calls that we used. Overall, the Shared DFA provides us with a way to amortize the cost of the DFA over multiple instances, since it is used only as a read-only lookup table.

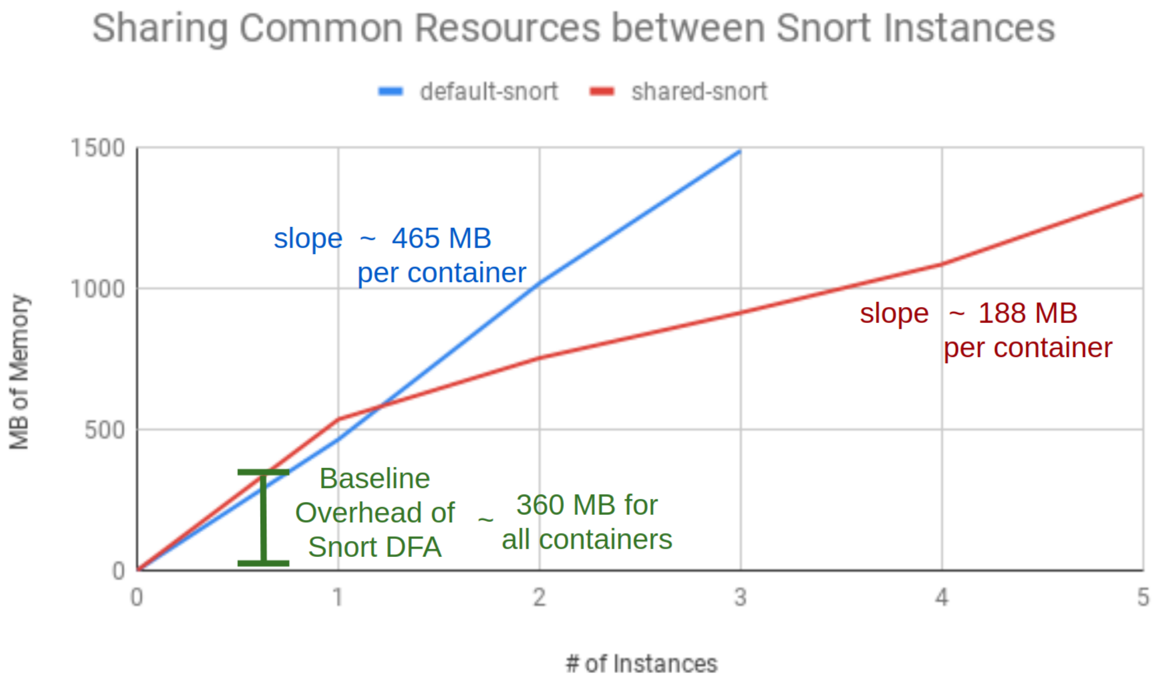

Figure 4.3: This graph shows the effectiveness of sharing the DFA data structure across all Snort instances. The first Snort instance incurs the overhead of the shared DFA, while the rest of the snort instances use a reduced amount of memory.

4.2 Combined Optimization

We implemented each of these optimizations to run on a Raspberry Pi 3B+ with 1 GB RAM and 1 GB swap memory. We analyze the combined optimization using the device-specific rulesets, reduced socket buffers, and the shared DFA data structure with respect to memory and performance. We use a C executable that invokes `system` calls to start and stop containerized Snort instances, as well as log measurements. While measuring the memory footprint of Snort, we use `iperf` to simulate network traffic and mimic the normal behavior of Snort. We use a Snort configuration with 1697 rules, according to a Linux-based IoT device, when evaluating the combined optimization.

4.2.1 Memory Footprint

We can see the memory footprint of Snort using each optimization, as well as the combined optimizations' footprint, in Figure 4.4. Combining all three optimizations allows us to run a maximum of 47 Snort instances simultaneously, at least ten-fold gain compared to using Snort with the community ruleset for registered users (snapshot-29120) and a default DAQ buffer size of 128 MB. We observe an increased startup time as we spawn more Snort instances. Therefore, we constrain the time for the containerized Snort instance to start up, to within three minutes, after which we consider the system unusable. The maximum number of simultaneously running instances for each individual optimization, as well as their combined gain, are shown in Table 4.4.

Optimization	Max # Snort Instances Running
none (default)	3
reduced-buffer	4
shared-DFA	5
custom-ruleset	6
combined-optimizations	47

Table 4.4: This table shows the maximum number of Snort instances that can be run using each optimization, as well as all of them combined.

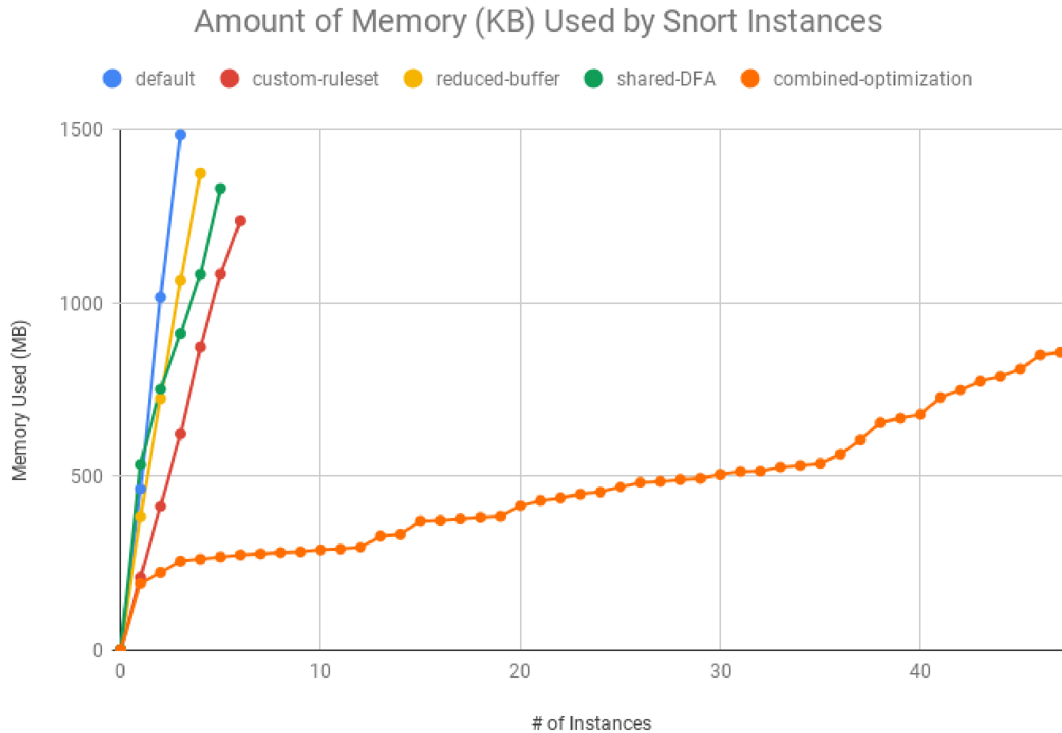

Figure 4.4: This graph shows the amount of memory used by multiple Snort instances for each optimization, as well as the combined footprint.

4.2.2 Performance

We measure the performance of each optimization, as well as the combined optimizations' performance by measuring the latency of packets processed by the Snort. Our results are shown in Figure 4.5 as a CDF plot that shows the percentage of packets that arrive under a certain latency. We exclude the latency of the first packet to reduce noise from ARP and other startup-induced costs. Based on the plot, the performance of the combined optimizations is even better than the default Snort, however this is slightly misleading. We hypothesize that this is due to the DFA being placed in shared memory, where it will not be evicted from the cache during a context switch. During a context switch in a process, the translation lookaside buffer (TLB) is flushed, causing memory private to that process to be flushed from the cache. The shared DFA is allocated in shared memory, therefore we believe that it remains in the cache through a context switch, resulting in faster lookup

times, and a reduced latency.

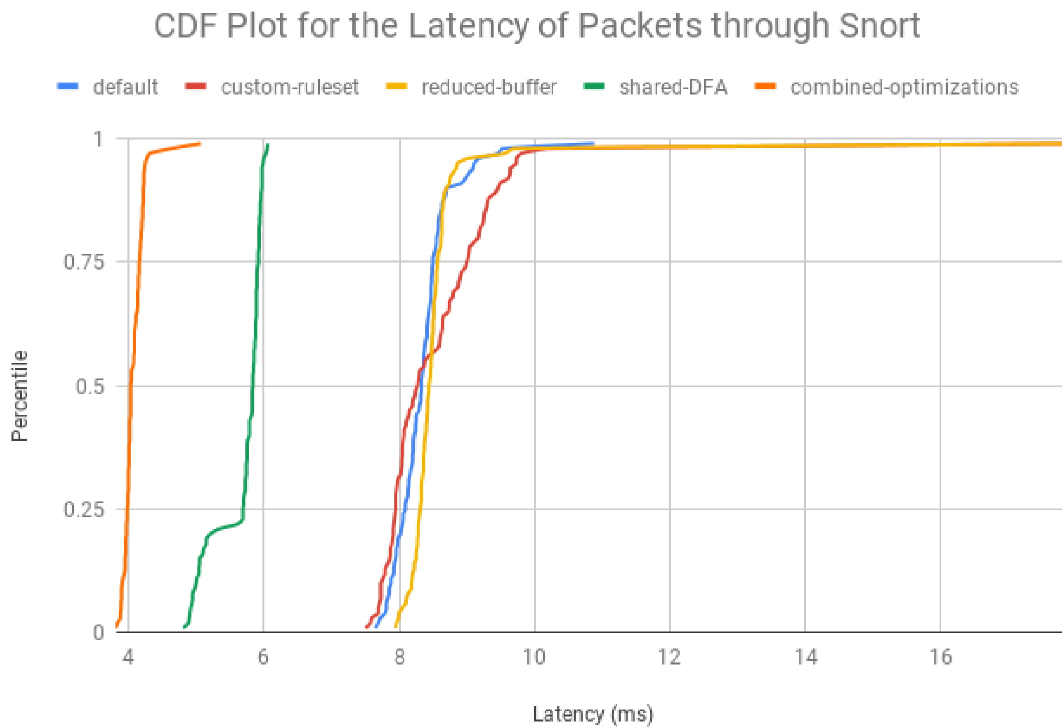

Figure 4.5: This graph shows the percentage of packets that arrive under a certain latency for each optimization, as well as all of them combined. We exclude the latency of the first packet to reduce noise from ARP and other startup-induced costs.

Chapter 5

Conclusions & Future Work

5.1 Conclusion

Through a few simple optimizations, we are able to increase the increase the number of Snort instances running on a Raspberry Pi 3B+ by at least ten-fold, helping move towards an easily-adoptable IoT security gateway for the common household. We identify the major memory contributors to the memory footprint of Snort, which are the heap memory used to match on the Snort rules, and the socket buffers used to store packet data. Snort's community rules are normally meant for a network-wide deployment and cannot all be applicable to a single IoT device. We are able to get at least a five-fold reduction on the size of the ruleset, for a majority of IoT devices, by removing unnecessary rules. The default buffer size (128 MB) is unnecessarily large for a single IoT device and can be reduced using runtime options. Finally, we share the DFA data structure as read-only data between Snort instances to amortize its cost over all instances. With these optimizations, we are able to increase the number of Snort instances running simultaneously on a Raspberry Pi 3B+ by ten-fold.

5.2 Contributions

This thesis's key contributions are the following:

- A set of tags for Snort's community ruleset for registered users (snapshot-29120) that are loaded into a `sqlite3` database and can be used with information about

the OS, device type, and manufacturer to create a reduced ruleset containing only rules that are applicable to a specific IoT device.

- A procedure for deploying Snort as a vNF instead of a single protector for the entire network.
- A modified implementation of Snort version 2.9.12 that allocates the DFA data structure on a shared heap, and can be accessed as read-only data by other instances of this modified Snort application.

5.3 Future Work

There are still a few more optimizations, with respect to memory, that could be explored to reduce the memory footprint of Snort and other vNFs. We explain our intuition on how they could be implemented and leave them for future work.

5.3.1 Specialized `malloc` Implementation

When sharing common resources in Snort, we dynamically allocate the DFA data structure in shared memory. We modify the snort source code by changing relevant malloc calls to shared malloc calls, that place the data structure on the shared heap. Unfortunately, when creating the data structure there are many deallocations on the shared heap as well, which leads to fragmentation of the heap. This is when a contiguous block of memory contains a mixture of allocated and unallocated segments. Once we grow the heap for memory allocations, it becomes difficult to shrink the heap because we are unable to easily determine which virtual pages are unused, if any. Therefore, using a malloc implementation that attempts to minimize the total heap size among the memory allocations/deallocations could further lower the memory footprint.

5.3.2 Anonymously Mapped Pages

The virtual memory breakdown shown in Figure 3.2, indicates around 8% of the memory footprint of Snort comes from anonymously mapped pages. These are allocations made from calls to `mmap`, using the `MAP_ANONYMOUS` flag, and could come from any part of the Snort application or its libraries. We were unable to easily determine where exactly these allocations are made and what they are used for. Further analysis methods may be

necessary to identify and optimize the allocation of these anonymously mapped pages, which contribute a non-trivial amount to the total memory footprint of Snort.

5.3.3 Automated Generation of Snort Rulesets

When creating device-specific rulesets, we utilize tags, described in Section 4.1.2, that characterize the applicability of each Snort rule. To utilize these tags and automatically create device-specific rulesets, we need to be able to automatically determine whether a tag is applicable to a certain device. A more detailed network discovery process will likely yield trimmer rulesets containing only relevant rules to that specific device. This automated scanning could also utilize Common Vulnerabilities and Exposures (CVEs) to create rules that are specific to an IoT device and the software running on it.

5.3.4 Descheduling Idle vNFs

IoT devices often have bursty traffic patterns rather than a constant flow of packets. This means that when there is no inbound/outbound traffic for an IoT device, its respective Snort instance may be waiting for packets and using RAM unnecessarily. An interesting idea posed in Zhang et al. [2016] is that we can use a hybrid of interrupt vectors and polling the network interface card to schedule an NF only when there are incoming/outgoing packets waiting to be processed. Instead of waiting for packets, the NF can deschedule itself, waiting to be signaled for when traffic needs to be processed. We can then suspend the state of any vNF, using a feature such as Docker checkpointing, and migrate the vNF to disk space to free up RAM when necessary. The main intuition is that we can take advantage of this hybrid polling process to run vNFs only with packets waiting to be processed, and save memory by suspending vNFs which have no packets to process. This memory optimization could induce additional latency, based on the time needed to migrate and resume a vNF from disk, and this tradeoff would need to be further analyzed.

5.3.5 Automated Stress Testing

Snort rulesets can grow to be very large and contain rules for many different protocols. When running experiments with live traffic, we test only a small subset of these rules to automatically determine if proper alerts are generated. Ideally, we would use a set of stress tests to determine that alerts are properly generated for every Snort rule. This would require a program to generate traffic that would be matched on by a given Snort rule. We

can then test the correctness of these optimizations, to ensure that we have not introduced any new bugs.

5.4 Final Thoughts

We aim to generalize the methods and solutions from this work, and apply them to other vNFs needed to secure IoT devices. Although not necessarily due to memory, datacenters also suffer from resource constraints when running many vNFs, specifically due to Snort and other IDS/IPS systems. The solutions posed in this work could be a step in the right direction to solving these resource constraints within datacenters as well. As technology advances, we may also notice that memory is not the bottleneck of the system anymore, requiring new methods for analysis and optimization.

All in all, we hope that the ideas posed in this paper help enable the use of vNFs to secure IoT devices, and make a positive impact towards an emerging, IoT-driven world.

Bibliography

Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai botnet. In *26th USENIX Security Symposium USENIX Security 17)*, pages 1093–1110, 2017. 1.1

Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016. 2.4.1

Simon Elias Bibri. *Smart Sustainable Cities of the Future*. Springer, 2018. 1.1

Thanh Bui. Analysis of Docker Security. *arXiv preprint arXiv:1501.02967*, 2015. 2.4.1

Cisco. Cisco Visual Networks Index. 2019. URL `https://www.cisco.com/c/m/en_us/solutions/service-provider/vni-forecast-highlights.html`. 3.1

Theo Combe, Antony Martin, and Roberto Di Pietro. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016. 2.4.1

Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016. 2.4.1

CVE. WannaCry CVE-2017-0144, 2017-0144. URL `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0144`. 1.1

CVE. SambaCry CVE-2017-7494, 2017-7494. URL `https://access.redhat.com/security/cve/cve-2017-7494`. 1.1, 2.5

Richard Cziva. *Towards Lightweight, Low-Latency Network Function Virtualisation at the Network Edge*. PhD thesis, University of Glasgow, 2018. 1.3, 2.6.3

Docker Inc. Docker, March 2013. URL https://www.docker.com/. 2.4.1

Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review*, 38(5):29–40, 2008. 2.5.2

R Guerzoni et al. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action, Introductory White Paper. In *SDN and OpenFlow World Congress*, volume 1, pages 5–7, 2012. 1.3

Sumi Helal and Christopher N Bull. From Smart Homes to Smart-Ready Homes and Communities. *Dementia and geriatric cognitive disorders*, 47(3):157–163, 2019. 1.1

Intel. Intel VTune Amplifier, 2017. URL https://software.intel.com/vtune. 3.2.1

Intel Corporation. Intel SGX, September 2013. URL https://software.intel.com/en-us/sgx. 2.4.1

Ronny Ko and James Mickens. DeadBolt: Securing IoT Deployments. In *Proceedings of the Applied Networking Research Workshop*, pages 50–57. ACM, 2018. 1.3, 2.6.1

Doug Lea. malloc/free/realloc written by Doug Lea, 2011. URL https://gist.github.com/ichenq/2166885. 3.3.4

Daniel Lezcano, Serge Hallyn, Stphane Graber, et al. Linux Containers (LxC), 2008. URL https://linuxcontainers.org/. 2.4

Linux. Free(1) man page, 2018. URL http://man7.org/linux/man-pages/man1/free.1.html. 3.2.1

Gordon Fyodor Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, USA, 2009. ISBN 0979958717, 9780979958717. 4.1.2

John Matherly. Complete guide to Shodan. *Shodan, LLC (2016-02-25)*, 2015. 1.1

Nick McKeown. Software-Defined Networking. *INFOCOM keynote talk*, 17(2):30–32, 2009. 2.2

Marc Norton. Optimizing Pattern Matching for Intrusion Detection. *Sourcefire, Inc.*, 2004. 2.5.2

Gerald J Popek and Robert P Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, 1974. 2.3

Raspberry Pi. Raspberry Pi 3 Model B+, 2018. URL `https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/`. 3

Martin Roesch et al. Snort: Lightweight Intrusion Detection for Networks. In *Lisa*, volume 99, pages 229–238, 1999. 1, 1.4, 2.5

Bhagya Nathali Silva, Murad Khan, and Kijun Han. Towards Sustainable Smart Cities: A Review of Trends, Architectures, Components, and Open Challenges in Smart Cities. *Sustainable Cities and Society*, 38:697–713, 2018. 1.1

Nate Stiller. MAC Address, OUI, and IAB Lookup, 2019. URL `https://www.macvendorlookup.com/`. 4.1.2

The Snort Project. Snort Community Ruleset for Registered Users, 2017. URL `https://www.snort.org/downloads/registered/snortrules-snapshot-29120.tar.gz`. 3.3.2

The Snort Project. Snort Community Ruleset for Registered Users, Snapshot-29120, 2019a. URL `https://www.snort.org/downloads/registered/snortrules-snapshot-29120.tar.gz`. 2.5.1

The Snort Project. Snort users manual 2.9.13. 2019b. URL `https://www.snort.org/documents/snort-users-manual`. 3.3.3

The Wireshark Team. Wireshark, 2019. URL `https://www.wireshark.org/`. 4.3

VirusTotal. Virustotal-Free Online Virus, Malware and URL scanner. *Online: https://www. virustotal. com/en*, 2012. 4.1.2

Ruozhou Yu, Guoliang Xue, Vishnu Teja Kilari, and Xiang Zhang. Network Function Virtualization in the Multi-Tenant Cloud. *IEEE Network*, 29(3):42–47, 2015a. 2.1

Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a Trillion (Unfixable) Flaws on a Billion Devices: Rethinking Network Security for the Internet-of-Things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, pages 5:1–5:7, New York, NY, USA, 2015b. ACM. ISBN 978-1-4503-4047-2. doi: 10.1145/2834050.2834095. URL `http://doi.acm.org/10.1145/2834050.2834095`. 1.2

Tianlong Yu, Seyed Kaveh Fayaz, Michael P. Collins, Vyas Sekar, and Srinivasan Seshan. PSI: Precise Security Instrumentation for Enterprise Networks. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017. URL `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/psi-precise-security-instrumentation-enterprise-networks/`. 1.3, 2.6.2

Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 3–17, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4292-6. doi: 10.1145/2999572.2999602. URL `http://doi.acm.org/10.1145/2999572.2999602`. 5.3.4