Aegis: Lightweight, Trusted “Bolt-on” Security Gateway for IoT Deployments


*Carnegie Mellon University - CyLab
{mmccorm1, sanjayc, guyue, tianlony, vsekar}@andrew.cmu.edu
†Carnegie Mellon Software Engineering Institute
{avasudevan}@sei.cmu.edu

Abstract—Many deployed IoT devices struggle to implement security best practices. Prior work identified a pragmatic solution of “bolting on security at the network” to secure these deployed IoT devices with device-specific protections. However, it is challenging to realize these solutions on low-cost platforms because the diversity of devices create an array of security needs and the inability to verify if per-packet security protections were applied. This paper is a practical solution to these problems. We realize a lightweight, trusted security gateway for IoT devices on a low-cost platform (Raspberry Pi 3B). To achieve this, we identify resource bottlenecks in a commodity network function and adapt it to a lightweight, device-specific operating mode. This enables a 10x increase in the number of network functions that can run concurrently. Additionally, we leverage micro-hypervisor based isolation and attestation to ensure packets were processed by the appropriate network function. Finally, we demonstrate the applicability of this gateway to current IoT deployments.

Index Terms—IoT, Network Security, SDN, NFV

I. INTRODUCTION

Diverse new devices are continually being adopted in our networks. These devices are enticing targets for attackers [1]–[5]. A survey of 45 home IoT devices identified a total of 39 known vulnerabilities [6]. As these vulnerable devices proliferate throughout our networks, there is a need to both protect the new devices from attackers (e.g., [1]) and to ensure that these new devices do not become pivot points allowing attackers to penetrate further into the network (e.g., [4], [5]).

Industry and the research community have proposed securing deployed IoT devices by “bolting-on security” in the network. This allows for protecting devices that might not support patching (e.g., a device vendor goes out of business) or are unable to run host-based defenses (e.g., antivirus software). These proposals have utilized a security gateway architecture for mediating an IoT device’s network access [7]–[11].

Most commercial options appear to apply a single set of monolithic security functions to all network traffic, whereas, recent research advocates for custom security profiles for each IoT device [9]–[11]. These custom profiles generate a unique, device-specific set of security functions for each device. These security functions are independent and isolated from each other, providing performance isolation (e.g., no collateral damage from other device’s traffic) and simplifying management of each IoT device’s security protections (e.g., clearly distinguishing the security protections for each device).

As the number of deployed IoT devices is predicted to continue growing exponentially, it is critical that an on-site security gateway be able to support a large number of devices. However, current security gateways that provide custom security profiles for each device are limited in their scalability on low-cost hardware, particularly when considering commodity network security functions (e.g., firewall, intrusion detection, proxy, etc.). They cannot simultaneously run >10 instances of commodity security functions on low-cost hardware (i.e., hardware costs comparable to those of IoT devices), as there is insufficient RAM for all of the security functions to exist simultaneously (Table I). The use of commodity security functions is crucial for advancing network security in the IoT realm, as it enables the direct transfer of existing best practices. Thus, lightweight versions of commodity network security functions are required for IoT security gateways.

Furthermore, in prior work the entire security gateway was in the trusted computing base (Table I). This assumes an attacker cannot tamper with any of the security gateway’s functionality. If an IoT device’s network traffic were sent to the incorrect security function, or skipped all together, an attacker could defeat the protections provided by the gateway. Similarly, an attacker could defeat the protections by modifying network packets after they are processed by a security function. Thus, a trusted gateway needs to provide a guarantee that an IoT device’s network traffic is processed by the correct security function and that only those security function can modify that traffic. Prior work is summarized in Table I, with a more detailed discussion in §II-C.

TABLE I: Comparison of security gateway systems.

<table>
<thead>
<tr>
<th>System</th>
<th>Cost</th>
<th>Bolt-on</th>
<th>Scalable</th>
<th>Custom profiles</th>
<th>Trusted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traffic Profilers</td>
<td>$150</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Smart Firewalls</td>
<td>$199</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Security Manager [7]</td>
<td>~$100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>IDIoT [8]</td>
<td>$10</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>IoTSec [9], [10]</td>
<td>~$1k</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Aegis</td>
<td>$35</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The goal of Aegis is to demonstrate a lightweight, trusted security gateway system on low-cost hardware. To achieve

1Aegis comes from Greek mythology and refers to the shield of Zeus.
lightweight security functions, we leverage domain specific insights (e.g., IoT device’s limited network requirements) for configuring the security functions and the potential for common security requirements (e.g., malware signatures) to be shared between multiple instances of a security function. These lightweight functions enable the use of custom security profiles. These are preferred over monolithic security functions, as the security functions in custom profiles are isolated inside virtualized network functions (VNF), to reduce the likelihood of policy interference by being more precise, and to simplify management by separately describing each device’s security policy.

To achieve trust, we employ a micro-hypervisor based security architecture [12]–[14]. A micro-hypervisor implements a software reference monitor to provide isolation and resource mediation [12], [14]. In addition, a micro-hypervisor also facilitates platform attestation whereby a certified software state can be communicated to remote entities [15]. Last but not least, a micro-hypervisor embodies a low trusted computing base that can support unmodified commodity OS and applications [12], [13].

Our trust architecture consists of two key components housed within the micro-hypervisor framework: (a) VNF configuration attestation, and (b) VNF packet signing. We employ remote attestation combining a platform TPM [15], [16] along with the micro-hypervisor to verify that the correct VNFs are running (§V-C). We leverage micro-hypervisor facilitated memory protections to isolate a packet signing agent from the remainder of the untrusted software, which then creates a digital signature to create connection between the packet, security function, and internal routing on the security gateway (§V-D). The packet signing agent thereby ensures that the packets are processed by the correct VNF. The packet signing agent also leverages micro-hypervisor assisted code whitelisting [17], [18] to ensure that the packet signing agent can only be invoked by authorized code (e.g., VNF) from within the untrusted OS and applications.

We demonstrate a practical realization of a security gateway that is scalable and trusted while operating on low-cost hardware, a Raspberry Pi 3B. Specifically, we show how we were able to reduce the memory requirement of the most widely deployed intrusion prevention system (Snort IPS) by 416 MB, allowing the security gateway to support 56 simultaneous instances (previously it was limited to three). Additionally, we increase the trust of the system by providing a guarantee that packets were processed by the appropriate security function. We demonstrate the utility of this prototype implementation on two use cases: commodity IoT in a home and networked 3D printers in a makerspace. We make the following contributions:

- **Lightweight:** Analyzing Snort IPS and modifying it to create a lightweight version for supporting device-specific operations. We demonstrate reducing the additional instance cost by a factor of 12.
- **Trusted:** A micro-hypervisor based security architecture to ensure that: (a) VNF is of the correct type and configuration (e.g., rule set), and (b) packets from the IoT device are routed to the correct VNF, and their contents are only altered by this VNF.

- **Prototype:** End-to-end system implementation on the Raspberry Pi 3 platform running commodity software, demonstrating device-specific security protections of commodity IoT devices.

## II. BACKGROUND AND MOTIVATION

### A. Motivating Examples

Researchers have identified many security vulnerabilities in deployed IoT devices [6]. We present two common IoT vulnerabilities to highlight the need for “bolting-on” security in the network to protect deployed IoT devices.

- **Default credentials:** An IoT device shipped with hardcoded admin password, that cannot be changed (a key vulnerability used by Mirai [1]). This allows an attacker to remotely login and execute arbitrary code. This could be mitigated by adding an additional layer of authentication in the network (e.g., an authenticating proxy) prior to accessing the IoT device, limiting the default credentials’ exposure (à la [9]).

- **Out-of-date software:** Maintaining the appropriate software version and “patches” on the specific IoT device is a tedious task, especially when considering a plethora of IoT devices. Prior to a patch being released, an IoT device might be vulnerable to a known exploit [19]. A potential solution is to add an Intrusion Prevention System (IPS) that drops packets matching the exploit’s signature, “patching” the IoT device in the network until the device’s software can be updated.

### B. IoT Security Gateways

Traditional network security solutions are limited in their ability to protect IoT devices. Host-based security solutions (e.g., antivirus) cannot be run on all IoT devices due to a lack of installation mechanisms or insufficient computational resources. Today’s network-based defenses are predominantly employed on the network periphery, leaving the devices vulnerable to attacks originating inside the local network (e.g., insider attacks). Furthermore, many of these network defenses are implemented in hardware middleboxes that are difficult to modify with new device-unique security requirements.

The network shows promise for securing these devices [9]–[11]. The network is a common link between most IoT devices and has an optimal view, able to monitor all of a device’s communications. Our work focuses on devices using an IP-based network. While some devices use other protocols (e.g., BLE, ZigBee, etc.) many connect to a hub on an IP network.

- **Need for scalable custom profiles:** Prior work has proposed securing deployed IoT devices by “bolting-on network security” using a security gateway collocated with the IoT devices [7]–[9], [11]. Traditionally, network security has been implemented using monolithic security functions to process all of the network traffic. However, these suffer from insufficient isolation and context [10]. This is magnified by the diversity of IoT devices, running a wide array of network services, and the anticipated exponential growth in the number of deployed IoT devices. Researchers have proposed using custom security


profiles (i.e., isolated, device-specific security functions) where a virtual switch routes packets to the appropriate security function (e.g., firewall, IPS, etc.), we refer to these as VNFs [9]–[11]. A comparison between a monolithic security function and device-specific security functions is illustrated in Fig. 1.

Fig. 1: Comparison of a security gateways implementing monolithic security functions with custom profiles.

Custom profiles provide context, logical isolation, and performance isolation [10], [20]. Context ensures that each device’s specific security vulnerabilities are addressed. Logical isolation stops differing policies from conflicting with each other (e.g., blocking the telnet port for the entire subnet when one device, with a dynamically assigned IP address, requires telnet for operation). Performance isolation ensures that traffic to one device does not adversely impact the processing of another device’s network traffic (as they are being processed separately). Additionally, this structure simplifies verifying the appropriate security actions are being applied to each device (a complex task as the number of devices increases for a monolithic security function). These advantages are ideal for IoT deployments, as the diversity of devices creates a many unique needs (e.g., one device needs an authenticating proxy, while another an IPS for a network path). Due to these advantages, we focus on implementing a security gateway that provides custom profiles, where each device has its own security functions.

1) Need for a trusted security gateway

The security gateway may itself be attacked to mitigate the protections it is providing to deployed IoT devices.

Modify security function: An attacker could modify the security function’s configuration (e.g., remove a vulnerability signature from an IPS). This could be detected by remotely attesting the VNF’s configuration and checking if it matches a known good state.

Packets skip security function: An attacker could cause a packet to not be processed by a VNF. This could be accomplished by either (a) changing the packet routing configuration or (b) modifying the data in the packet after the packet leaves the security function.

C. Prior Work

Security gateways can be grouped based upon how their security functions are implemented.

Monolithic Security Functions: Current commercial products implement monolithic security functions. A baseline set of generic security properties can be applied to all network traffic. These commercial products can be grouped into two broad categories based upon the monolithic VNFs they provide.

- Traffic Proﬁlers (e.g., DOJO): Snoop on network trafﬁc to collect behavioral characteristics from metadata. These characteristics are compiled from multiple users in order to detect anomalous behavior.
- Smart Firewalls (e.g., CUJO, Bitdefender Box2, RATtrap, Norton CORE): Mediate a device’s network access (some as a wireless access point). While marketed as smart firewalls, they often perform additional security functions (e.g., deep packet inspection).

Similarly, research efforts have applied monolithic security functions to protecting IoT devices. These have followed two primary goals: (a) patch management or (b) limiting network connectivity (e.g., only allowing an IoT device to connect to a certain set of cloud servers). Work on patch management recommended using an in-hub security gateway to aided patching (e.g., applying patches when the device is not regularly in use) [7]. Works on limiting a device’s network connectivity, add rules to a monolithic security function (e.g., iptables) after passively monitor a device [8]. These monolithic security functions are limited as they do not provide isolation and become complex to manage as the number of devices protected increases.

Custom Profiles: Recent research artifacts on security gateways have proposed protecting IoT devices using custom profiles [9], [11]. These security gateways intercept an IoT device’s network traffic and perform device-specific security operations (e.g., deep packet inspection, firewall, etc.) on this traffic. They tailor the security applied to a device’s specific vulnerabilities by applying per-device security patches in the network. Additionally, some of these works advocate for additional hardware (e.g., TPM [11]) or software APIs in future IoT devices.

Limitations: Solutions for securing existing IoT devices have been limited in their capabilities or assumed the availability of significant computational resources. Some only limit outbound network traffic (e.g., only allowing an IoT device to access its cloud server) [8], [21]. This potentially misses malicious actions/activities within the local network (e.g., a malicious insider). More robust solutions have assumed access to significant computing resources (not found in most home networks) and thus ignored questions of scalability on a lightweight platform [9], [10].

Prior works, summarized in Table I, showcased the security benefits with a proof of concept implementation; they only provided limited details on their deployment considerations (e.g., number of IoT devices supportable, latency increase, etc.). Additionally, they implicitly trust the security gateway, assuming an attacker cannot tamper with its operations.

As IoT devices begin using end-to-end encryption, it limits security functions to only being able to analyze packet headers/metadata. This is a limitation of existing systems as well as Aegis. However, complementary work has looked at trusted ways of granting access to the cleartext data [22], [23].

III. SYSTEM ARCHITECTURE

Our approach to achieving a lightweight, trusted security gateway is primarily geared to mitigate threats from a network attacker. This applies to both inbound and outbound traffic
our high-level security property is stated below.

inbound and outbound traffic goes through a security gateway, a known vulnerability were checked for the vulnerability’s appropriate security function (e.g., packets to a device with the right type and with the specified configuration (e.g., rule set), customized to an IoT device, is running on the gateway.

We want to provide each IoT device with its own device-specific security protections. The average US household has \( \sim 8 \) connected devices [24], this is anticipated to double in the next 5 years. Thus, our scalability goal is to support 20 IoT devices concurrently, each with a unique VNF.

Given a packet \( \text{pkt} \), generated outside the security gateway, that is originated from or destined to an IoT device \( \text{dev}_i \), the resulting packet \( \text{pkt}' \) processed by the gateway can be denoted as:

\[
\text{pkt}' = \text{VNF}_i(\text{pkt}), \quad \text{where} \quad \text{VNF}_i = \{\text{Type}_i, \text{Config}_i\}
\]

This high-level property implies two sub-properties:

- **P1:** VNF instance validation. A VNF of the correct type and with the specified configuration (e.g., rule set), customized to an IoT device, is running on the gateway.
- **P2:** Packet path and data validation. The packet processed by the virtual switch has been routed to the correct VNF, and its contents have only been altered by this VNF.

### B. Challenges

Designing and implementing a security gateway with our goals of scalability and trust while realizing our desired security properties is met with a number of non-trivial challenges. The attacker can exploit the vulnerable IoT devices. Specifically, an attacker can create a known exploit for the IoT device, with a goal of either (1) sending malicious payloads (e.g., a known exploit) or (2) making the device inoperable (e.g., DoS attack). The attacker does not have physical access to the gateway or IoT devices. Additionally, the attacker can attack the gateway in order to exploit the vulnerable IoT devices. Specifically, an attacker can compromise VNFs running on the security gateway (e.g., change the rule set running on the VNF). An attacker can modify how the virtual switch routes packets on the security gateway (e.g., have packets not go to the appropriate VNF). Additionally, an attacker can modify the data in a packet while it is in transit to or from a VNF.

### C. Threat Model and System Assumptions

**Attacker Capabilities:** Our attacker has network access to IoT devices with known vulnerabilities. The attacker can create and modify network traffic (e.g., sending packets containing a known exploit for the IoT device), with a goal of either (1) sending malicious payloads (e.g., a known exploit) or (2) making the device inoperable (e.g., DoS attack). The attacker does not have physical access to the gateway or IoT devices. Additionally, the attacker can attack the gateway in order to exploit the vulnerable IoT devices. Specifically, an attacker can compromise VNFs running on the security gateway (e.g., change the rule set running on the VNF). An attacker can modify how the virtual switch routes packets on the security gateway (e.g., have packets not go to the appropriate VNF). Additionally, an attacker can modify the data in a packet while it is in transit to or from a VNF.

### Table II: Potential network “patches” for IoT vulnerabilities.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Potential Solution</th>
<th>Security Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default credentials</td>
<td>Authenticating proxy</td>
<td>Squid</td>
</tr>
<tr>
<td>Unpatched vulnerability</td>
<td>IPS</td>
<td>Snort</td>
</tr>
<tr>
<td>Exposed, unused services</td>
<td>Firewall</td>
<td>iptables</td>
</tr>
<tr>
<td>No encryption</td>
<td>VPN</td>
<td>OpenVPN</td>
</tr>
<tr>
<td>DoS susceptibility</td>
<td>Rate limiting</td>
<td></td>
</tr>
</tbody>
</table>
While the attacker can compromise the gateway, we limit the attacker to not performing the following attacks: (1) dropping packets, (2) modifying packet data at the network interface controller (NIC), and (3) stealthy malware (e.g., attack then revert to a benign state).

System Assumptions: We assume that all traffic to the IoT device goes through the security gateway. We are not considering an attacker that modifies the network to directly access the IoT device (e.g., spoofing the WiFi AP). Similar to other security gateways, Aegis does not identify attacks sent over an encrypted channel (e.g., known exploits transmitted over TLS), but can identify anomalous behavior in the metadata (e.g., sudden large traffic volume in a DoS attack). However, complementary works show promise for analyzing encrypted traffic [22], [23]. We assume the following software components are in our trusted computing base (TCB) on the gateway: the micro-hypervisor, the virtual switch, and the VNF. This is already an order of magnitude smaller than existing approaches that trust the entire software stack on the gateway (Table I). We discuss potential paths to further reduce our TCB in §VIII.

D. Design Considerations

We discuss two main architecture design decisions: the gateway’s physical location and its hardware platform.

Location: The VNFs could easily be provisioned and run in the cloud. However, this architecture potentially limits security coverage and impacts performance. A cloud-based gateway creates the potential for local traffic to bypass the security gateway. Additionally, it delays the application of “network patches” (e.g., data not encrypted until leaving the cloud). Thus, we chose an on-site gateway.

Low-cost Platform: Ideally, the security gateway would run on a wireless access point. To emulate a high-end wireless access point, we selected the Raspberry Pi 3B for its low-cost and comparable hardware specifications.

E. Architecture Overview

Similar to the approaches that push security into the network using SDN [10], we want to have every device’s first-hop to be the security gateway (as shown in Fig. 2). This provides coverage on both internal and external traffic. However, unlike prior approaches, our primary focus is on making the gateway implementation lightweight and trustworthy.

Data plane: The dataplane of the security gateway operates on low-cost hardware in the same location as the deployed IoT devices. Security is provided using VNFs (e.g., Firewall, Intrusion Prevention System, etc.) running on the security gateway to “patch” specific vulnerabilities (e.g., a VNF could drop packets to an unused, open port on an IoT device, drop traffic matching a known vulnerability, etc.). Each IoT device has a unique VNF, specific to its security needs, that provides performance isolation and simplifies management. Once network traffic arrives at the security gateway, it is sent to an appropriate VNF by a virtual switch. Packets exiting a VNF are verified to have been processed by the appropriate VNF. This is accomplished on low-cost hardware while meeting IoT deployment needs (e.g., minimal latency increase).

Control Plane: A remote SDN controller dynamically configures and monitors the security gateway. A high-level policy specifies the VNF for each IoT device. This controller verifies the configuration of the dataplane (i.e., running VNF instances and routing configuration) every epoch (e.g., every hour).

Verifiable and Secure Routing: Realizing our security properties (P1 & P2 from §III-A) requires certain foundational capabilities such as component isolation, resource mediation, and platform attestation. We leverage a micro-hypervisor based system security architecture [12]–[14] that supports the aforementioned capabilities on commodity unmodified OS and system software, while characterizing a low trusted computing base that is amenable to formal verification (to rule out potential bugs) [12], [13].

The micro-hypervisor in our architecture houses two key components: (a) VNF packet signing (accomplished by a protected packet signing agent), and (b) VNF configuration attestation. The micro-hypervisor memory protections isolate and protect the packet signing agent from the remainder of the untrusted platform software. The packet signing agent then creates a digital signature to create connection between the packet data, security function and internal routing on the security gateway (Fig. 4; §V-D). The packet signing agent thereby ensures that the packets are processed by the correct VNF. The packet signing agent also employs code white-listing [17], [18] in order to ensure that the packet signing agent can only be invoked by the virtual switch. Finally, we employ remote attestation combining a platform TPM [15], [16] along with the micro-hypervisor in order to verify that the correct VNFs are running on the platform (§V-C).

IV. REALIZING LIGHTWEIGHT VNF INSTANCES

Aegis aims to provide each device it protects a unique VNF, this enables isolation and customization. Although similar ideas have been proposed by prior work, their implementations are based on commodity servers with ample hardware resources (e.g., 48 GB memory) [10], [26], [27]. It is challenging to implement this on a low-cost platform with limited hardware resources (e.g., 1 GB memory) and the increasing number of deployed IoT devices.

A naïve solution is to build customized VNFs from scratch, minimizing the functionality to achieve a lightweight implementation. However, this approach could easily miss important security features and hardly leverages the existing best practices. Therefore, we choose to use commodity network security functions; however, running these large VNFs with limited resources requires resource usage profiling, bottleneck identification, and implementation optimizations.

The main bottleneck limiting scalability was the memory required by each instance of a the security function. In a sampling of security functions, the memory required spanned from 1 MB to >452 MB per instance (reference Fig. ??). This creates a scalability challenge as a resource constrained
platform such as the Raspberry Pi 3 can only run four of the largest security functions simultaneously, while the average US household has eight connected devices [24].

An intrusion prevention system (IPS) instance is likely required for each IoT device Aegis is protecting. We analyzed the most popular security function (Snort) to identify potential methods for making existing security functions more lightweight in terms of memory required. We believe that this analysis approach is applicable to other security functions, Snort was selected as an example because it is the most widely deployed intrusion prevention system in the world.

A. Challenge

Running multiple instances of the default configuration of Snort, with the full community rule set, is limited by the platform’s available RAM. Most of the low-cost platforms (e.g., OpenWRT routers) that we analyzed had <2 GB of RAM. This bottleneck limited the number of simultaneous instances of Snort that could be running to <6. Assuming that every device being protected by the security gateway requires its own instance of Snort, this would limit a single hardware platform to supporting at most six IoT devices. We want to include all of the community rules that are applicable to an IoT device to provide broad coverage and benefit from the large support community publishing generic attack signatures. Using only a single Snort instance to process every device’s traffic removes the benefits of isolation and context, potentially resulting in allowing attacks to succeed or performance degradation.

Analysis Process: We analyzed Snort to identify the most significant contributors to its memory consumption. Using a two-phased analysis approach: first analyzing the configuration inputs, and second analyzing the source code. This approach allowed us to identify memory reductions that could be achieved without modifying the existing security function (and without access to source code), and what further improvements are possible using minor source code modifications.

We analyzed the source code with Intel’s VTune Amplifier profiling tool. This allowed us to identify major memory consumers. For Snort, we identified that the majority of the memory was allocated on the heap for rules and their processing, followed by socket buffers (as shown in Fig. 3).

![Virtual memory profiling of a VNF running Snort with different configurations: Default (full community rule set), Configuration reductions (reduced socket buffer and rule set), and all optimizations.](image)

Fig. 3: Virtual memory profiling of a VNF running Snort with different configurations: Default (full community rule set), Configuration reductions (reduced socket buffer and rule set), and all optimizations.

Configuration Analysis: Most network security functions have a large number of configuration options. Their default use assumes that a single instance is going to be running for analyzing all traffic on a network of multiple devices. For example, Snort is designed to analyze 200-500 Mbps of network traffic generated by multiple devices on the same subnet. This is different from our use case, where each IoT device has its own instance of Snort. A single device will generate significantly less network traffic than the aggregate of all the devices on a network. Thus, there is likely a storage buffer that can be reduced based upon our use case.

The full community rule set, composed of 10,918 rules, is designed to protect a wide array of devices. Using custom profiles allows us to reduce these rules to only the rules that apply to that particular device (e.g., an IoT device running Linux does not need Windows rules). Furthermore, we noted that this reduced rule set would likely be common to many IoT devices, requiring that each Snort instance allocate memory for these rules, with only a small fraction being device specific. Thus, further memory reductions could be gained if rules were only placed in memory once.

B. Optimizations

Based on our analysis, we implemented a set of optimizations to make a lightweight version of Snort. We group our optimizations into two categories: (1) modifying the configuration, and (2) modifying the source code. Configuration Modifications: Based upon our memory usage profiling, we identified two configuration items that would have the greatest impact on Snort’s memory consumption: (1) socket buffer size and (2) the number of rules.

Socket Buffer. Snort’s socket buffer is set by default at 128 MB, which with other metadata and a 4 KB data block size results in an allocation of 165.5 MB. As the peak bandwidth of many IoT devices is <20 Mbps\(^2\) (equivalent to a <2.5 MB socket buffer). Thus the socket buffer could be safely reduced from 128 MB to 3 MB, resulting in a reduction of 162 MB per Snort instance.

Number of rules. The community rule set contains 10,918 rules. The amount of memory required by a given Snort instance is directly proportional to the number of rules it is configured with (~27.5 KB per rule), as shown in Fig. ??.

We developed an automated tool that categorizes a rule set (based upon rule descriptions and exploit references) into 116 separate categories to more precisely identify which rules might be applicable for a given IoT device. Once a rule set is tagged, it can easily be queried to identify those that are applicable to a given device (e.g., based upon an Nmap scan). We sampled IoT devices and applied only the rules that had tags applicable to the device and realized up to a five-fold decrease in Snort rules (shown in Table III).

<table>
<thead>
<tr>
<th>Device</th>
<th>Relevant Community Rules</th>
<th>Memory Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux NAS</td>
<td>2,205 (20.2%)</td>
<td>~234 MB</td>
</tr>
<tr>
<td>D-Link Camera</td>
<td>1,693 (15.5%)</td>
<td>~248 MB</td>
</tr>
</tbody>
</table>

\(^2\) The majority of IoT devices sampled had a peak throughput of <1 Mbps.
Source Code Modifications: With access to the source code, we can further optimize Snort to be more lightweight. Eliminating unused rules created significant reductions in required memory. However, we envision that a substantial portion of rules will be common across multiple devices. Thus, we will have the same rule in memory multiple times (i.e., once for each IoT device being protected). Our insight is that if we place these common rules into a shared memory region, we can only have one instance of the rules in memory and greatly increase the number of simultaneous security functions that we can have simultaneously instantiated.

The initial security function does not experience any memory usage reductions; however, subsequent instances can be instantiated at a significant reduction in required memory (e.g., 30 MB per instance). Using these combined optimizations, we were able to reduce the required memory footprint of Snort by more than 12x per instance, (as shown in Table IV). These optimizations allow a single hardware platform to support more than double our goal.

TABLE IV: Memory required for additional Snort instances.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Memory per Additional Instance</th>
<th>Max Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>452 MB</td>
<td>33</td>
</tr>
<tr>
<td>Reduced Socket Buffers</td>
<td>323 MB</td>
<td>5</td>
</tr>
<tr>
<td>Reduced Rules</td>
<td>201 MB</td>
<td>6</td>
</tr>
<tr>
<td>All Config Reductions</td>
<td>68 MB</td>
<td>25</td>
</tr>
<tr>
<td>All optimizations</td>
<td>29 MB</td>
<td>56</td>
</tr>
</tbody>
</table>

V. TRUSTED PACKET PROCESSING

As described before (§III-A), the two security properties that Aegis enforces are: (a) P1: VNF instance validation – the correct VNF for an IoT device is running, and (b) P2: Packet path and data validation – the packet processed by the virtual switch has been routed to the correct VNF. We would like to identify a departure from these properties in a timely manner.

A. Strawman Solutions

Multiple methods could be used for generating trust in the security gateway’s operations.

Known-good Provisioning: The security gateway can be loaded with a known good software stack to start with and all the software components can be trusted during runtime. Unfortunately, this scheme offers little or no protection against an adversary who can mount runtime network attacks and exploit vulnerabilities on a software stack that exposes a wide attack surface such as a system that runs a commodity OS and other applications. Trusted Platform Module (TPM): Alternatively, a hardware root of trust could be added. This trusted piece of hardware could be used for performing remote attestations of the network functions running on the security gateway. However, a TPM on its own is limited, as an attacker could maliciously reset and input data into the TPM, resulting in the TPM reporting an incorrect attestation measurement. Additionally, a TPM could provide little insight if an attacker were modifying packets after being processed by the VNF.

B. Our Solution

Designing and implementing a system that provides our desired properties P1 and P2 (See §III-A), is nuanced since reasoning about correct forwarding paths is non-trivial. We propose a micro-hypervisor based system security architecture towards trusted packet processing. This approach offers a nice middle ground between completely trusting all the software and using only dedicated hardware. Further, the micro-hypervisor architecture provides us with several foundational capabilities: mediation, isolation and attestation. In combination, these allow us to achieve our trust properties.

We accomplish property P1 (§III-A) using a TPM with access to the TPM mediated and protected by a hypervisor. This allows the gateway controller to remotely attest the gateways configuration (i.e., running VNF instances), and protects the attestation measurements from being tampered with by an attacker. For property P2 (§III-A), we utilize per-packet signing, where access to the signing keys are isolated and protected by the hypervisor. This ensures that an attacker modifying packets or attempting to send them to the incorrect VNF will be identified before the packet is allowed to leave the security gateway. This combination of trust tools (e.g., hypervisor and TPM) provides increased robustness. We now discuss these aforementioned mechanisms in more detail.

C. Remotely Attest Configuration

Our system architecture includes a controller that knows and directs the configuration of the dataplane on the security gateway. The controller specifies the VNF instances⁴ that should be running. If an attacker on the gateway is able to modify the VNF instance, they could remove attack signatures and allow a network attacker to successfully exploit an IoT device being protected by the security gateway.

To identify this type of an attack, where the dataplane configuration has been modified, we employ remote attestation. Specifically, we have a TPM on the security gateway that periodically measures each running VNF instance (e.g., computing a SHA1 of the VNF’s executable). These measurements are stored in the TPM’s platform configuration registers (PCRs) and sent to the gateway controller as a quote (i.e., a signed message containing current PCR values). The controller can compare the PCR values in the quote with its known value for that measurement. This allows the controller to identify if the VNF instance has been modified (and indicates that an attacker has modified the security gateway). However, as it is only computed periodically, during the window between measurements, the VNF could be in a compromised state. Performing this remote attestation for each running VNF allows for isolating which devices might be under attack/compromised (compared to only knowing there was an attack).

Access to the TPM is mediated by the hypervisor. This can be performed for both access to a physical TPM and a virtual TPM. A physical TPM provides a hardware root of trust. However, this hardware is not standard on low-cost platforms.

⁴We are defining VNF instance as the combination of the security function (e.g., Snort) and its configuration (e.g., the Snort rule set).
and its performance is limited by the data bus it is physically connected to. Alternatively, a virtual TPM can be instantiated in the micro-hypervisor. While it cannot provide a physical root of trust, it does not have the same performance limitations as a physical TPM (e.g., limited data transfer rate by the SPI data bus). Additionally, it is not limited in storage based upon physical configuration but can be configured in software (e.g., adding additional PCR registers). As a microbenchmark, we compared the time required to store a measurement (e.g., extend a PCR) on a physical TPM with a virtual TPM. As shown in Table V, the virtual TPM was 20x faster while providing 8x more measurement storage locations (PCRs).

TABLE V: TPM PCR extend comparison.

<table>
<thead>
<tr>
<th>TPM</th>
<th>Median Time</th>
<th>PCR Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical</td>
<td>17.2 milliseconds</td>
<td>24</td>
</tr>
<tr>
<td>Virtual</td>
<td>0.78 milliseconds</td>
<td>configurable, up to 200</td>
</tr>
</tbody>
</table>

D. Packet Signing

To confirm that packets are sent to and processed by the correct VNF we added a digital signature to each packet. The digital signature relies on a shared secret key between the virtual switch and the VNF. The digital signature is verified and then discarded; thus there is no modification required for the security function or the IoT device being protected. A high-level overview of this process is shown in Fig. 4. If the virtual switch receives a packet without a proper signature from a VNF, the packet is dropped.

Packet Signature Implementation: We want to sign and verify each packet twice, once prior to being processed by a VNF and once after being processed by a VNF. This allows a VNF to modify a packet (e.g., encrypt the data) while still being able to confirm both that the packet was intended for the VNF and that the VNF processed the packet. To ensure that an attacker cannot maliciously generate signed packets, we want to protect the signing key with the micro-hypervisor, only allowing certain processes to calculate a packet’s signature. As both the cryptographic operations of signing a packet and the system calls of invoking the micro-hypervisor are costly, we microbenchmarked each operation to understand the performance cost and noted that the total packet signing overhead is ~1 millisecond (544 µseconds for HMAC and 472 µseconds for hypervisor call).

VI. SYSTEM IMPLEMENTATION

We implemented the Aegis dataplane on a Raspberry Pi 3B, configured to host a wireless access point for IoT devices. Each IoT device has a unique security policy that is specified by an administrator to protect each device from known vulnerabilities. This security policy is given to the security gateway controller. A single controller can manage multiple security gateway dataplanes.

IoT device operations are not modified, the security gateway transparently applies the security functions specified in the security policy. The default action is to deny network access to devices without a security policy. Upon initial connection, the controller configures the dataplane. Subsequent packets are processed on the dataplane without any controller actions.
A. Components

The controller is implemented on a separate physical machine running Ubuntu and the OpenDaylight SDN controller. This setup emulates a remote controller on a cloud provider’s infrastructure (e.g., AWS). We extended OpenDaylight to add the ability to instantiate our VNFs based upon a provided security policy and verify the received attestation measurements from the dataplane.

The dataplane is implemented on a Raspberry Pi 3B running Raspbian. The wireless access point is created using hostapd. Additionally, it has a wired connection that connects it to the local network, which we assume later connects to the internet.5

We used uberXMHF [25], an open-source⁵, formally verified, commodity micro-hypervisor framework to implement our trust architecture. The micro-hypervisor provides fundamental security primitives such as secure boot, memory isolation and attestation. It also allows development of micro-hypervisor extensions (uberapps) that can build upon the core micro-hypervisor in a modular and integrity-protected manner. We leveraged an already existing uberapp, a virtual TPM, to support remote attestation of the running VNFs. In addition, we developed a new packet signing uberapp for securely signing and verifying each packet’s digital signature.

Packet routing to VNFs is performed using a virtual switch running openvswitch (OVS). This switch was configured to allow the controller to specify routing rules remotely on the security gateway. We extended OVS’s userspace switch to add the ability to sign and verify each packet, with our uberapp.

The VNFs are implemented using Docker containers, where each VNF runs a set of security functions (e.g., IDS, firewall, etc.) as well as a module for signing and verifying packets. We chose to place multiple security functions inside a single container to reduce the number of keys maintained by the virtual switch. Furthermore, having a chain of VNFs would require the signature incorporate additional data for verify packet’s path through the VNF chain.

To demonstrate the functionality of the security gateway, we implemented the following VNFs: (1) Snort IPS to block known vulnerabilities, (2) Squid authenticating HTTP proxy to protect devices with default credentials, (3) iptables rate limiting to limit an attacker’s ability to launch a DoS attack on an IoT device. Each VNF was built from a base Ubuntu container with the minimal dependencies to run the required security function. Each VNF was configured with two virtual NICs, both connected to the virtual switch. Additionally, each VNF included a module for verifying incoming packets and signing outgoing packets. This module was implemented using NFQUEUE to send packets to the sign/verify function which would call the uberapp on the micro-hypervisor.

Snort IPS: Snort was configured with the optimizations discussed in §IV. Device-specific vulnerability signatures were added to create a network patches for vulnerable devices (e.g., SambaCry signature for a NAS).

Squid Authenticating HTTP Proxy: Squid was configured to authenticate HTTP traffic. This additional layer of authentication can patch devices with hard-coded credentials.

Rate Limiting: Iptables was used to limit the number of connections and bandwidth an IoT device could utilize. This both protects the device from being the victim and from generating DoS attacks.

B. Operational Description

Aegis’s operation is broken into two phases: first when a device initially connects to the gateway, and second its steady-state packet processing flow.

Upon an IoT device initially connecting to the security gateway, the security gateway will contact the gateway controller to determine how to configure the dataplane. If the device is found in the controller’s security policy, it will configure the dataplane with appropriate VNF and packet routing rules. Otherwise it will have the gateway drop packets from that device (default drop). Once the dataplane is configured, the controller will remotely attest the VNF.

All subsequent packets from the device will be processed on the dataplane without any required involvement from the controller. However, the controller will periodically perform a remote attestation of the running VNFs to verify they have not been modified.

Each packet arriving at the gateway will be sent to the virtual switch. The switch will add a digital signature to the packet. For our proof-of-concept we added this signature into the packet payload; however, the packet could also be encapsulated with the signature added to the encapsulation headers (e.g., using a network service header). Next, the switch will route the packet to the appropriate VNF. The VNF will verify that the packet’s signature using a symmetric key it shares with the virtual switch. The VNF will perform the security functions, sign the packet, and send it back to the switch. The switch will verify this new signature and route the packet out of the gateway to its end host.

VII. System Evaluation

We benchmark Aegis to determine the maximum number of Snort instances it could run, as well as the maximum throughput. After these stress tests, we measured its performance in IoT deployment settings.

A. Scalability

Applying the optimizations discussed in §IV, we achieved an increase of >10x the number of simultaneous Snort instances running on a Raspberry Pi 3B, shown in Fig. 5a. Additional Snort VNFs were instantiated until an out of memory error occurred. The memory utilized was measured with the free command. Reducing the configurations enabled an 8x increase, and utilizing shared memory enabled an 18x increase in the number of simultaneous Snort instances.

To evaluate the performance of these Snort instances, we measured the latency by performing 100 HTTP GET requests to a simple web server using httping, shown in Fig. 5b. As anticipated, the reduced configurations had similar latency to

5 Additional wired interfaces can be added using USB-Ethernet adapters.
6https://uberxmhf.org
the baseline. Moving the attack signature rules into shared memory reduced the latency by 2.7 to 4.3 milliseconds on average (30%-50% reduction). We hypothesize that this latency reduction is from the shared memory not being evicted from the cache during context switches.

B. Trusted Computing Base

As described previously in §III-C our trusted computing base (TCB) consists of the micro-hypervisor, VNF and the virtual switch. We add a small amount of code, 85 source lines (corresponding to our packet signing uberapp), to the already small TCB (≈5500 source lines [25]) of the uberXMHF micro-hypervisor code base. This keeps the micro-hypervisor code-base amenable to (future) formal verification as demonstrated by uberXMHF’s x86 platform verification effort [28].

C. Packet Signing Measurements

Generating a digital signature for each packet comes at a performance cost in terms of reduced throughput and increased latency. We measured the latency and throughput using netperf (e.g., cryptographic operations) for a wired connection between two linux computers, as this would note the greatest impact on latency (as there is significantly more latency inherent on a wireless channel). To measure only the performance costs of packet signing, we sent traffic through a single VNF that performed no security functions. We used OVS in user mode with no VNFs as a baseline (OVS User) for comparison to performing the packet signing in userspace (Signature OVS & VNF) and performing a hypercall where the signing operation is performed by the micro-hypervisor (Hypervisor Signature). The performance impacts are shown in Fig. 6, where we noted a reduction in throughput of 13% for packet signing and 67% for having the micro-hypervisor protect the packet signing. Similarly, we noted an increase in latency of 18% and 43%.

In order to determine how much of an impact these performance costs would have in the IoT domain, we sampled a number of commodity IoT devices. To understand their network requirements during normal user interaction, we measured their median throughput to approximate network utilization and their inter-packet arrival time to give an insight on latency tolerance (shown in Fig. 7). We noted that many devices have low throughput (<200Kbps) and that only the camera had a majority of its packets arrive less than 1 millisecond after previous packets. For the non-camera devices the packet inter-arrival times can be loosely grouped into two categories: (a) sending data chunk (<250 μseconds) accounting for <30% of packets and (b) after a delay (>30 milliseconds) accounting for >70%. Thus, many IoT devices have low network throughput and are often send in bursts, a 10 Mbps security gateway could likely support 20 devices. Additionally, a latency increase of 650 μseconds would only impact a small percentage of many IoT device’s traffic.

D. Use Case Examples

To demonstrate Aegis, we deployed it in two simulated environments, a smart home and a 3D printing makerspace.

Simulated Home IoT Deployment: Our simulated home IoT deployment containing two devices: a camera with default credentials and a NAS with unpatched vulnerability (e.g., SambaCry). The camera’s traffic was routed through an authenticating HTTP proxy, while the NAS’s traffic was routed through an IPS. To measure the impact of Aegis on this deployment, we used httping to measure the latency of 500 HTTP GET requests to the NAS (shown in Fig. ??). Going through the gateway added 3.48 milliseconds, the IPS processing packets added an additional 4.33 and milliseconds, and packet signing further increased the latency by 3.69 milliseconds.

3D Printing Makerspace: Our simulated makerspace was composed of multiple types of networked 3D printers. Each of these 3D printers had different DOS vulnerabilities, and were protected by a uniquely configured rate limiting VNF. We evaluated Aegis in this deployment by measuring the median time for sending a 1 MB file to the 3D printer. We noted an increase of 765 milliseconds (processing by the VNF without packet signing increased latency by 307 milliseconds).

VIII. DISCUSSION

We discuss future optimizations for Aegis that would make it more lightweight and trusted.

Increasing Packet Processing Performance: Our implementation routed packets in userspace by our virtual switch. This is known to provide poor performance (we measured a 60% reduction in throughput compared with a kernel space virtual switch), thus virtual switches have a kernel space drivers or utilize direct transfer of packets from a NIC to userspace (e.g., DPDK). The sign/verify functionality could be moved into the virtual switch’s kernel driver to provide significant
performance improvements, as tests with the kernel driver and only the VNF performing sign/verify actions only showed a throughput reduction of 11%.

**Broadening Scalability:** The current architecture requires all stateful VNFs (e.g., IPS) remain in memory once initiated for a device. However, this requirement could be removed by adapting the system to utilize Checkpoint/Restore in Userspace (CRIU). As many IoT devices have bursty traffic, the VNF is only required to be in memory when traffic is being generated. During periods of inactivity, the VNF state could be saved to disk and restarted once another burst of traffic is received. This functionality is similar to how a virtual switch can drop routing rules if no traffic matching that rule has been seen over a specific time period. This would allow for protecting an increased number of devices on a given deployment. There would be a small latency cost each time a device restarted communicating over the network when the checkpointed VNF is restarted (similar to the latency for starting a new VNF).

Additionally, we believe our approach to making Snort lightweight could be applied to other VNFs. In particular, large buffers could be reduced as they are not needed for IoT device’s low throughput. VNFs that share common configuration items, similar to IPS rules, could be modified to share these.

**Strengthening Trust Properties:** Our current threat model assumes that we can trust the VNF and the virtual switch in addition to the micro-hypervisor. While the micro-hypervisor framework we employed has been formally verified for memory isolation and integrity properties [28], VNF and virtual switch add a considerable order of complexity in comparison. We identify two approaches for further reducing the size of our TCB while strengthening the types of properties we can achieve: (1) adding an additional layer of sign/verifying at the NIC kernel driver, (2) placing the NIC driver inside the hypervisor. Adding an additional layer of packet signing would require an attacker to have a kernel exploit in order to tamper with packets. This could be further improved by placing the NIC driver inside the hypervisor, allowing the hypervisor to mediate all access to the NIC. With this approach we would no longer need to trust the virtual switch and the VNF components, while being able to achieve even stronger properties (e.g., any packet sent to the gateway is processed by the corresponding VNF).

**Increasing Hardware Capabilities:** IoT devices currently have low network utilization, many sending <3 Mbps (reference Fig. 7). In the future, these requirements are likely to increase; however, more capable hardware platforms will also be available at low-cost. For example, currently the Raspberry Pi 3B is limited by its network hardware capabilities (i.e., 100 Mbps NIC). The Raspberry Pi 4 has removed this bottleneck, providing a 1 Gbps NIC (in addition to increased CPU and memory). This change in hardware moves the limiting bottleneck to the CPU, as the network throughput of Aegis on a Raspberry Pi 4 is dependent upon the complexity of the operation being performed on the network traffic (with differences of >200 Mbps).

**IX. Conclusion**

We demonstrated an implementation of Aegis, a lightweight, trusted “bolt-on” security gateway for IoT deployments. Using a low-cost platform, we were able to support deployments of up to 56 IoT devices where each device has its own lightweight version of a commodity VNF. Additionally, we were able to provide a trust guarantee that every packet arriving at the virtual switch was processed by the appropriate VNF with a per-packet latency increase of 645 µseconds (where only devices sending continuous data, such as a camera, sent the majority of their packets at a higher rate), demonstrating the feasibility of a low-cost platform for providing custom security profiles and trust that the appropriate security functions are being utilized for securing small IoT deployments.

**REFERENCES**


[17] A. Vasudevan et al., “Lockdown: Towards a safe and


[22] D. Naylor et al., “And then there were more: secure communication for more than two parties,” in *CoNEXT 17*.


