JETFIRE: A Low-Cost, Trusted IoT Security Gateway

Matt McCormack, Amit Vasudevan, Guyue Liu, Tianlong Yu, Sanjay Chandrasekaran, Brian Singer, Sebastián Echeverría, Grace Lewis, Vyas Sekar

December 1, 2020

CMU-CyLab-20-002

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

JETFIRE*: A Low-Cost, Trusted IoT Security Gateway

Matt McCormack,* Amit Vasudevan,† Guyue Liu,* Tianlong Yu,* Sanjay Chandrasekaran,*
Brian Singer,* Sebastián Echeverría,† Grace Lewis,† Vyas Sekar*
*Carnegie Mellon University - CyLab, †Carnegie Mellon Software Engineering Institute

Abstract

Many studies have pointed out security problems with IoT deployments. Given the diversity of devices and the lack of concerted efforts from device manufacturers to adopt best practices, recent efforts have recommended pragmatic "bolt on" security gateways at the network layer to secure IoT deployments using software-defined principles. While such gateways are an attractive option, they raise two natural concerns: (1) *Can the gateway architecture be trusted?* and (2) *Can we deliver these benefits to low-cost deployments?*

This paper presents JETFIRE, a practical, low-cost system with built-in trust for software-defined security gateways. In designing and implementing JETFIRE, we make three key contributions: (1) A practical and deployable basis for trust using a micro-hypervisor root-of-trust; (2) A scalable low-cost system design and implementation to support fine-grained per-device policies; and (3) A formal analysis of the protection JETFIRE offers against infrastructure threats by construction. We demonstrate that JETFIRE provides intrinsic security against a broad spectrum of known attacks against such software-defined architectures. We also show that JETFIRE offers security at low cost; e.g., a $35 Raspberry Pi can effectively support custom per-device IPS instances for a small IoT deployment of 50+ devices. We also show an end-to-end validation of JETFIRE on a representative home IoT deployment.

1 Introduction

Internet of Things (IoT) devices are continually being added to our networks (e.g., homes, factories, etc.). Unfortunately, IoT devices are plagued by vulnerabilities [3, 44, 106]. Attackers have used these devices as stepping stones to attack protected enterprise networks [51, 59], and to compromise critical facilities such as city-scale infrastructures [23, 52] and smart factories [36, 43]. These compromised devices have also been used as launch pads for other attacks [5, 70].

Unfortunately, IoT device manufacturers continue to have poor security practices and the devices often lack management interfaces to be patched and/or configured. Thus, to satisfy the imminent need to protect these IoT devices from attackers' malicious actions (e.g., [5, 43]) and to ensure that these devices do not become pivot points for larger attacks, industry and academic efforts have proposed securing IoT deployments with on-site network gateways [8, 9, 17, 35, 40, 65, 78, 106].

These security gateways use software-defined principles to intercept traffic to and from an IoT device and apply network layer security protections via *software middleboxes* (e.g., a firewall) [35, 105]. These middleboxes implement network-layer protections tailored to each individual IoT device's vulnerabilities. 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). The specific protections are determined by a policy agent running on a centralized controller managing the gateway.

Ideally, a software-defined IoT security gateway must satisfy two fundamental requirements:

- *Trusted:* Gateways entail a single point of failure and this raises a paradoxical question: *Is the system providing these network-layer protections trustworthy?* This concern is not merely hypothetical; prior work has demonstrated attacks against the controller [95, 103], the control channel [37, 95, 103], and middleboxes [56, 86].
- *Low-Cost:* We want a low-cost solution that can be integrated into existing IoT ecosystems by run commodity software on existing hardware. At the same time, we need sufficient scalability to support constantly growing IoT deployments and extensibility where new protections can be added for adapting to new threats.

Unfortunately, existing approaches fail to achieve one or more of these of requirements [56, 86, 105]. For example, secure enclave-based approaches (e.g., SGX, TrustZone) and other trusted hardware (e.g., TPM-based solutions) provides a root-of-trust, but imposes a hardware requirement that may not be present on all platforms (e.g., restricted to specific processors) and incur significant challenges to adding new capabilities (e.g., requiring new hardware). We presently lack a low-cost, end-to-end approach that can be deployed on a broad class of existing commodity hardware.[1]

In this paper, we present the design and implementation of JETFIRE, a low-cost, trusted security gateway. We address key design and implementation challenges in adding cost-efficient and extensible trust to a software-defined security architecture. First, we use formal modeling to systematically identify key loci for applying relevant protections that enable trusting a gateway's packet processing. Second, we adopt a *micro-hypervisor-based system architecture* as it provides a root-of-trust that supports integrating protections in commodity software across a broad hardware base (cost-effective) and rapidly adding new protections (extensible) [4, 88, 89]. Third, as determined by our formal model, we use key micro-hypervisor provided capabilities such as *attestation* to verify software instances, *mediation* to enforce correct packet routing, and *isolation* to prohibit

*JETFIRE is an autobot from the fictional Transformers universe, who leverages commodity technology to mitigate evil. We adopt this fitting name for our low-cost, trusted IoT security gateway.

[1] A recent workshop paper by Mccormack *et al.*, [45] presented a high-level vision; their work is conceptual and falls short of a concrete implementation.

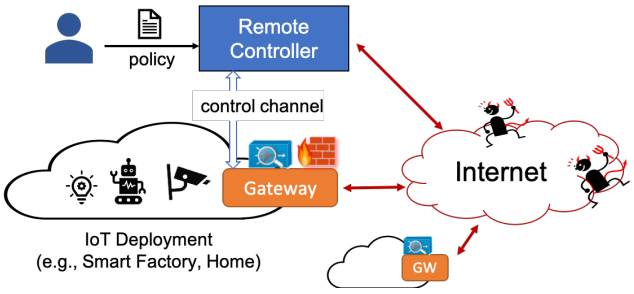


Fig. 1: Software-defined IoT security gateway architecture, the controller's security policy directs provisioning device-specific middleboxes to process each IoT device's network traffic on a local security gateway.

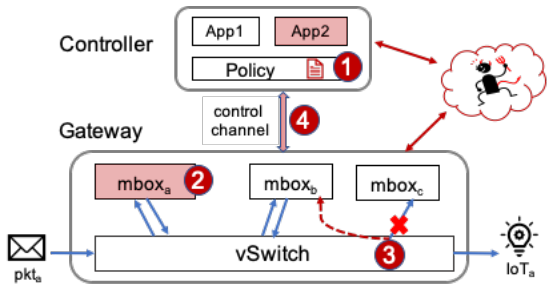


Fig. 2: Example attacks against current IoT security gateway architectures altering a gateway's packet processing.

tampering. We integrate these protections into the software processing packets on the gateway and controller to guarantee that all output packets are processed by the correct middlebox in a known good state. Finally, we enable fine-grained per-device security policies on low-cost platforms, by reducing the footprint of canonical security middleboxes such as intrusion prevention systems (IPS).

We implement an end-to-end system, JETFIRE, on a low-cost Raspberry Pi and demonstrate its ability to patch real IoT vulnerabilities in a representative smart home deployment. We evaluate the security of JETFIRE using a model-driven analysis and show that it provides security by construction, in addition to mitigating 10 published attacks against software-defined architectures. In terms of performance and cost, we demonstrate how a single Pi instance can offer device-specific IPS instances to protect small-to-medium scale deployments of 50+ IoT devices, with minimal impact on network performance.

Contributions and Roadmap:

- A practical micro-hypervisor based design of a trusted software-defined IoT security gateway that implements fine-grained protections guaranteeing output packets are processed by the correct middlebox (§3 and §5).
- An approach for reducing a middlebox's footprint to create lightweight, device-specific versions (§6).
- A formal model that identifies salient architectural components and a comprehensive security evaluation that demonstrates that JETFIRE provides protections, by construction, against a broad class of attacks including 10 published attacks on software-defined network architectures (§4 and §8).
- A practical demonstration of our system on a realistic IoT deployment showing performant and scalable network-level protections for commodity IoT devices (§9).

We plan to release our open source code base [15] to allow others to build on top of this trusted and low-cost architecture.

2 Background and Motivation

We start by discussing current IoT security gateway architectures (§2.1). Then, we motivate the need for a trustworthy gateway by presenting example real-world attacks (§2.2).

2.1 IoT Security Gateways

Traditional security solutions include host-based (e.g., antivirus) and network-based approaches (e.g., firewalls). For vulnerable IoT devices, host-based approaches are ineffective due to a lack of standard programming stacks and limited device resources [106]. Thus, defenders must use network-based approaches to protect their IoT deployments.

A *software-defined* gateway architecture [8, 9, 17, 35, 65, 78, 105, 106] has been proposed to secure IoT deployments. At a high level, the gateway intercepts all network traffic to and from an IoT device and runs virtualized middleboxes (e.g., firewall) to impose a security policy (e.g., block IoT devices from starting ssh connections). Compared to traditional static network defenses with baked-in policies, a software-defined architecture uses a *centralized controller* to flexibly define and configure customized policies.

Figure 1 shows an example IoT deployment protected by a software-defined security architecture. The controller configures the gateway via the control channel. For example, if a smart light is found to have an unpatched backdoor [1], the controller could initialize a firewall on the gateway to block access. The number and type of middleboxes depends on the defense strategy. A weak defense strategy could run a few shared middleboxes for all IoT devices [8, 78], while a stronger defense strategy may deploy one middlebox *per IoT device* [35, 105, 106]. A virtual switch (vSwitch [55]) routes packets to the appropriate middlebox, with the controller dynamically configuring the vSwitch's routing rules.

2.2 Need for Trustworthy Gateways

While software-defined gateways are promising for securing IoT deployments, it becomes *ineffective* when the architecture itself is *under attack*. We substantiate and motivate the need for trustworthy IoT security gateways by discussing four concrete, real-world attacks against such gateways, as identified by prior work [37, 45, 56, 86, 95, 103] and limitations of existing piecemeal solutions to mitigate all of these attacks.

As shown in Fig. 2, the controller (①), components on the gateway (②,③), and interaction between the controller and the gateway (④) can be exploited. While we discuss each attack separately, it is possible for a single attack to consist of a combination of these example attacks.

Table 1: Prior gateways and piecemeal security solutions.

Project	Root of trust	Mitigates Attacks A1	A2	A3	A4
Existing IoT Gateways [8, 35, 78, 105, 106]	OS	N	N	N	N
Trustworthy Middleboxes [24, 56, 71, 86]	SGX, TrustZone	N	Y	N	N
Secure Controllers [29, 57, 76]	OS	N	N	N	N
Secure Protocols [21, 34, 38, 39, 48]	OS + Crypto	N	N	Y	Y
JETFIRE (Our system)	Micro-hypervisor + Crypto	Y	Y	Y	Y

Attack 1 - Tamper with security policy(A1): The first attack (① in Fig. 2) targets the controller's security policy (à la [95]). An attacker can gain access to the controller, in a manner similar to how they gained access to the gateway in Attack 1. Once on the controller, the attacker can modify the IoT device's security policy. For example, specifying a more lenient middlebox (e.g., changing an IPS to a firewall), which allows the attacker's exploit to transit the gateway without detection.

Attack 2 - Alter middlebox operations (A2): The second example attack (② in Fig. 2) targets modifying the middlebox (à la [56, 86]). An attacker can gain access to the gateway, using credentials (e.g., from a data breach) or an unpatched vulnerability (e.g., [50, 102]). Once on the gateway, the attacker can modify the middlebox's configuration and remove rules that block an IoT device's known vulnerability. Now the attacker's exploit can bypass the security gateway's middlebox protections and compromise the IoT device.

Attack 3 - Alter packet path (A3): The third example attack (③ in Fig. 2) targets a packet's path on the gateway (à la [45]). An attacker that has compromised the OS could modify the packet's header to cause it to be routed to the incorrect middlebox, which fails to block the exploit payload.

Attack 4 - Inject malicious control channel messages (A4): The fourth example attack (④ in Fig. 2) targets the control channel (à la [37, 45, 95, 103]). In practice, a secure control channel (e.g., TLS) is not often used [74, 100]. An insecure channel allows an attacker to inject malicious messages. For example, sending openflow commands to reconfigure the vSwitch such that packets bypass a middlebox, thereby allowing the attacker's exploit to pass through undetected.

Limitations of existing solutions: As shown in Table 1, current IoT gateways [8, 35, 78, 105, 106] do not secure any of the above attacks. While there is some prior work securing individual pieces of the architecture, they still lack end-to-end trust (see details in §11). First, recent work on trustworthy middleboxes uses trusted enclaves to run middleboxes inside untrusted cloud environments (e.g., [56, 86]), but this solution requires specific hardware (e.g., SGX [27], TrustZone [6], TPM [7]) which is not widely available. Second, research on securing the controller (e.g., [57, 76]) has been focusing on using permissions to limit the access of multiple applications, but cannot provide runtime protections against an attacker capable of compromising the OS. Finally, existing secure tunnels (e.g., IPSec, TLS) and work on customized verification protocols (e.g., [34, 38, 39, 48]) can be used to achieve traffic integrity, but they alone are not enough to defend against all attacks.

3 System Overview

In the previous section, we have made a case for a trustworthy gateway. For such a gateway to be deployed in practice across a wide range of IoT deployments, it must also be low-cost. In this section, we first define what we mean by trustworthy and low-cost (§3.1), then present JETFIRE's overall architecture (§3.2), assumptions(§3.3), and challenges(§3.4).

3.1 System Goals

Overarching Trust Property (G1): a trustworthy software-defined IoT security gateway should provide a guarantee that *all output packets are processed by the correct middlebox, even when under attack.* We formulate this guarantee in §4 and examine how it mitigates existing attacks in §8.2.

Achieving the overarching trust property (G1), requires a root-of-trust that current software-defined IoT security gateway approaches lack (Table 1). This root-of-trust must provide foundational capabilities (e.g., memory isolation) to enable building a *holistic defense* on both the data and control planes.

Low-Cost (G2): A IoT security gateway that meets the above trust requirements alone is not very useful, if it cannot be readily deployed within today's edge IoT ecosystems. We view small, localized IoT deployments such as homes, offices, and manufacturing floors as those are most likely to experimentally deploy IoT (often having less than 20 localized devices [10, 26, 63, 87]) and experience more frequent attacks [68]. However, they often have limited budgets for IoT security. A high-end solution would not be practical for these customers. We therefore focus on providing low-cost gateways for small to medium enterprises supporting upto 50 devices with a single hardware gateway costing less than $100.

3.2 Key Components

To meet these goals, we envision JETFIRE, a *trustworthy* and *low-cost* software-defined IoT security system, that enables a new trustworthy "security-as-a-service" offering that providers (e.g., ISPs, CDNs) can offer to IoT consumers.

In contrast to existing gateway architectures [35, 105, 106], JETFIRE runs both the controller and gateway software on top of a carefully chosen *root-of-trust* (§5.1).The root-of-trust provides capabilities for isolating sensitive data (e.g., control policy, secure keys) and attesting the integrity of running software (e.g., middleboxes), while supporting many hardware platforms and commodity software (e.g., Linux, Docker).

Building upon the root-of-trust, we add four extensions to achieve our overarching trust property (G1) while mitigating the attacks previously mentioned in § 2.2. (1) We migrate the control policy into the isolated memory protected by the root-of-trust (mitigating attack class ①). (2) We use a performant attestation approach to verify the integrity of running software

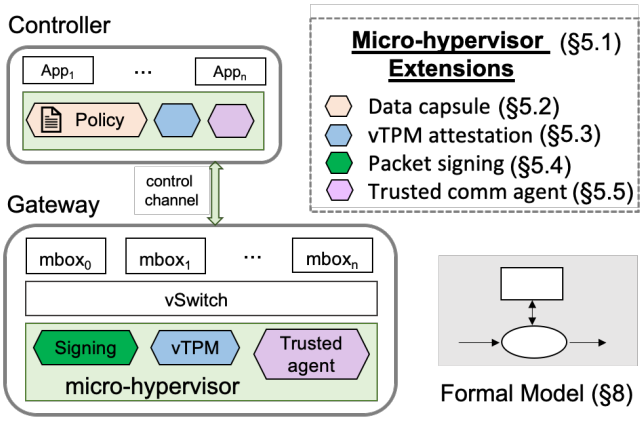


Fig. 3: JETFIRE's trusted IoT security gateway adds fine-grained protections to provide low-cost, end-to-end packet processing guarantees.

(mitigating attack class ②). (3) In each gateway, we design a trusted signing mechanism between the vSwitch and each middlebox to protect a packet's path and data (mitigating attack class ③). (4) We create a secure channel between the controller and the gateway to protect control messages (mitigating attack class ④). Our formal system model (§8.1.1) and comprehensive system evaluation (§8.2) shows that JETFIRE can achieve our trust property and mitigate several prior attacks.

3.3 Assumptions

System Assumptions: We assume all packets to and from an IoT device must go through the gateway as their *first-hop*. And we scope our system to only providing network protections to devices using an IP-based network. While some devices use other protocols (e.g., BLE, ZigBee), many use IP directly or connect to a hub on an IP network. We also assume middleboxes are correctly implemented and are able to block all network exploits targeting the IoT device they are protecting.

Threat Model: We consider a powerful network adversary capable of compromising the gateway and controller's operating system (OS) via the network. The adversary's aim is to render the gateway ineffective, and then access unprotected IoT devices. The adversary can flexibly choose combinations of attacks from the literature (e.g., attacks in §8.2), but cannot directly access an IoT device or use an evil twin attack (e.g., [67]) to bypass the gateway.

We do not aim to protect against an attacker generating DoS conditions (e.g., maliciously dropping packets [103]), nor provide confidentiality to packets and middleboxes (e.g., [56, 86]). Attacks modifying the controller's global network view by impersonating an IoT device or advertising false paths (e.g., [18, 95, 103]) are out-of-scope of this work.

3.4 Challenges

We highlight two challenges towards achieving our goals.

- **Good performance with a small TCB (§5)** For a trustworthy IoT security gateway to be used in practice, it must provide good performance. Additionally, the added trust should not come at the expense of a large TCB. Our key idea is to use a combination of isolation and attestation techniques so that we can isolate small pieces of critical software while attesting bigger, legacy software components.
- **Scalable middleboxes on low-cost platforms (§6)** A low-cost gateway needs to support continuing growth IoT devices ([10, 26, 63, 87]). However, existing low-cost platforms (e.g., Raspberry Pi) only supports 3 IPS middleboxes (see details in §9.2), which is not enough for a normal household (having on average 8 connected devices [13]). We identify that memory consumption is the main bottleneck for scalability and propose two optimization techniques for scaling support to approximately 50 IoT devices.

4 Formulate Trust Requirements

We begin by formulating our overarching trust property. We create a formal model of today's software-defined IoT security gateways (§2.1) to inform the design of JETFIRE (see discussion and evaluation of this model in §8). This model helps us define our overarching trust property (§3.1) and derive four required sub-properties protecting critical components and interfaces. These properties then define JETFIRE's trust requirements for achieving our overarching trust property.

Overarching Trust Property: Given a network where all of an IoT device's inbound and outbound traffic goes through our trusted security gateway, *GW*, our goal is to ensure that any packet, *pkt*, output by the gateway was processed by the correct middlebox while operating in a known state, so that benign packets are allowed and malicious packets are dropped. Our trust goal can be denoted as:

$$\begin{aligned} \forall pkt \in BenignPkt,\ processPkt(GW, pkt) = \text{Allow} \\ \forall pkt \in MaliciousPkt,\ processPkt(GW, pkt) = \text{Drop} \end{aligned} \tag{1}$$

To achieve this goal in the presence of an attacker, the entire gateway architecture must be trustworthy, expressed formally:

$$\begin{aligned} TrustedGateway(GW, Controller) \iff \\ tamperProof(policy) \wedge \\ correctInstance(vSwitch, mbox_{0,\ldots,n}, apps_{ctl}) \wedge \\ secureChannel(channel_{gw}, channel_{ctl}) \wedge \\ \forall mbox_i,\ authenticateRoute(vSwitch, mbox_i) \end{aligned} \tag{2}$$

$$\text{Where: } GW = \{vSwitch, \{mbox_0, \ldots, mbox_n\}, channel_{gw}\}$$
$$Controller = \{policy, apps_{ctl}, channel_{ctl}\}$$

Our overarching security property is composed of four sub-properties, where each maps to a trustworthy requirement: *tamperProof* (TR_{sw1}), *correctInstance* (TR_{sw2}), *secureChannel* (TR_{comm1}), and *authenticateRoute* (TR_{comm2}).

Security Policy Isolation and Mediation (TR_{sw1}): The first sub-property is to protect the security policy stored in the controller. Controller applications are subject to attacks [95, 103] which make the security policy vulnerable. As the correctness of the rest of the system is based upon this policy, we need to ensure it is tamper proof. To achieve this, the security policy needs to be isolated in protected memory with all access

mediated by a trusted entity (e.g., blocking the OS and other untrusted applications from accessing the security policy). Our security policy isolation and mediation sub-property (TR_{sw1}) can be denoted as:

$$tamperProof(policy) \iff isolatedMemory(policy) \wedge mediatedAccess(policy) \quad (3)$$

Component Instance Validation (TR_{sw2}): Besides the security policy, the software of key components must not be altered by an attacker (e.g., Attack 2 where the middlebox was altered [56, 86]). Such alterations can be detected by validating key software components are running the correct instance. Software components that must be validated includes the controller application, vSwitch and all middleboxes. Our component instance validation sub-property (TR_{sw2}) can be denoted as:

$$correctInstance(vSwitch, mbox_{0,\ldots,n}, app_{ctl}) \iff remoteAttest(apps_{ctl}) \wedge remoteAttest(vSwitch) \wedge \forall mbox_i, remoteAttest(mbox_i) \quad (4)$$

Packet Path and Data Validation (TR_{comm1}): Each packet must be routed to the correct middlebox as specified by the security policy. Prior work on Internet routing has advocated for per-hop path authentication to validate that packets followed the specified path [39, 48]. We aim to provide similar guarantees in order to detect packets maliciously routed to the wrong middlebox (e.g., Attack 3). In particular, we need to verify whether the *intended path* of a packet has been enforced, and whether packet data has been modified in-between. We denote our packet path and data validation sub-property (TR_{comm1}) as:

$$\forall mbox_i, authenticateRoute(vswitch, mbox_i) \iff \forall pkt,\ intendedPath(pkt, policy) = mbox_i \implies actualPath(pkt) = vSwitch \rightsquigarrow mbox_i \rightsquigarrow vSwitch \wedge unmodifiedData(pkt, vSwitch, mbox_i) \quad (5)$$

Control Message Integrity and Authentication (TR_{comm2}): To protect against control channel attacks (e.g., Attack 4)[18, 37, 95, 100, 103], we aim to ensure that the control channel is secure. To achieve this, the control channel needs to be authenticated and encrypted so that data transmitted over the channel has not been modified or spoofed (e.g., only the controller can send middlebox configuration commands to the gateway). Meanwhile, the *secret keys* used by the channel need to be isolated and any access is mediated by a trusted entity, denoted:

$$secureChannel(channel_{gw}, channel_{ctl}) \iff authenticatedEncrypted(channel_{gw}, channel_{ctl}) \wedge isolatedMemory(keys) \wedge mediatedAccess(keys) \quad (6)$$

A system that provides these properties will be secure *by construction* and mitigate the example attacks in §2.2. Next, we design a system to provide these trust requirements.

5 System Design for Low-cost, Low-TCB Trust

Based on our formal model of today's software-defined IoT security gateways (§4), we first identify a low-cost, low-TCB root-of-trust that provides the foundational security capabilities (e.g., isolation) required to achieve our overarching trust property (§5.1). Then, we discuss our approach for building fine-grained security protections that realize each of our trust requirements (§4) on top of this root-of-trust.

Table 2: Comparison of root-of-trust design space options.

Root of Trust	Security Capabilities	Low-Cost	
		Legacy software	Hardware
Trusted enclave (SGX, TrustZone)	isolation, attestation	No system calls, Limited memory	Limited processors
Secure languages (Rust, OCaml)	isolation, mediation	Requires reimplementation	x86, ARM
Micro-hypervisor	isolation, mediation, attestation	Supports	x86, ARM, microcontroller

5.1 Micro-hypervisor as a root-of-trust

As current software-defined IoT security gateways lack a root-of-trust (Table 1), we begin by adding a root-of-trust. The root-of-trust must be available for low-cost hardware (e.g., ARM) and support legacy software without requiring reimplementation to be practical in the IoT domain.

Design Alternatives for Root-of-Trust: The design space for root-of-trust can be categorized along two axes (summarized in Table 2). First, hardware dependent approaches (e.g., SGX enclaves [56, 86]) have been used to secure middleboxes in cloud environments. Unfortunately, these hardware features are not common on low-cost platforms and only support limited applications (e.g., no system calls [27]). On the other hand, pure software approaches such as formal verification and secure programming languages too have limitations. However, these approaches are hard to directly deploy today as they require significant reimplementation and verification effort to add protections to existing software. As many commonly used software applications can span over 100,000 lines of C/Java, these approaches quickly become intractable.

Why a Micro-hypervisor: Rather than using a pure hardware or software approach, we advocate using a hybrid approach in the form of a *micro-hypervisor*. A micro-hypervisor [46, 77, 88, 89] is in essence a software reference monitor [69], that acts as a guardian of system resources (e.g., files, sockets). Hypervisors have been used to integrate fine-grained security protections into commodity software; e.g., identifying covert malware, providing trusted system calls, attestation, debugging, tracing, application-level integrity and confidentiality, trustworthy resource accounting, on-demand I/O isolation, trusted path, and authorization [11, 19, 22, 41, 42, 47, 61, 73, 75, 79, 83, 84, 92, 94, 96, 101, 104, 108, 112, 113]. A *micro-hypervisor* retains the foundational capabilities (isolation, mediation, attestation and extensibility) of traditional feature-rich hypervisors, but with a small software base that is amenable to formal verification to ensure it is implemented without vulnerabilities [4, 88, 89]. Further, they only require hardware support for virtualization, allowing it to run on most existing low-cost hardware platforms (e.g., ARM [90], x86 [77, 88], microcontroller [4]). It directly supports commodity software without

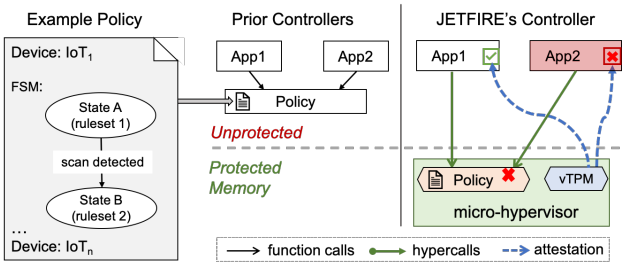


Fig. 4: JETFIRE migrates the policy file and processing logic into a data capsule, protected by the micro-hypervisor. An application (App2) that is not in the whitelist or fails at attestation cannot access it.

any limitations or modifications. Thus, a micro-hypervisor is a low-cost root-of-trust that is well suited to the IoT domain.

Unfortunately, a root-of-trust alone is insufficient for achieving our overarching trust property (§3.1) as it does not innately integrate with the existing gateway software to enforce specific protections. Next, we show how we build on top of the micro-hypervisor to achieve each of our trust requirements. To this end, we build four micro-hypervisor extensions (shown in Fig. 3): *data capsule* and *vTPM* for protecting the software (TR_{sw1}, TR_{sw2}), and *packet signing* and *trusted agent* for protecting the communications (TR_{comm1}, TR_{comm2}). We discuss each of these below.

5.2 Data Capsule for Security Policy Isolation and Mediation

Recall that our trust guarantee requires that the security policy be in isolated memory with mediated access. A naïve approach is to place the entire controller into memory protected by the micro-hypervisor. However, this approach creates two unwanted impacts: (1) it significantly *increases the TCB*, potentially exposing the micro-hypervisor to new attacks.[2] (2) it incurs *performance penalties* for all operations as every system call or external functionality needs to be mediated by the micro-hypervisor.

Our Approach: Instead of placing everything into the TCB, we carve out small critical pieces (e.g., control policy, secret keys) from the software, and migrate them into the micro-hypervisor as a *data capsule*. Each data capsule is isolated from unprotected memory and all accesses are meditated, providing *fine-grained* protection.

Figure 4 shows an example data capsule protecting the controller's security policy. The security policy specifies a finite state machine (FSM) for each IoT device. Each state of the FSM describes a specific middlebox configuration (e.g., Snort with ruleset 1). Transitions between states are based upon alert messages sent by the middlebox. For example, an alert message (or series) could indicate a network "scan detected", triggering the controller to reconfigure the middlebox with ruleset 2 which performs deep packet inspection.

In a traditional security gateway, the security policy along with other controller applications run in unprotected memory which is exposed to attackers (e.g.,① in Fig. 2). Rather than placing all of them (20k lines of code for OpenDaylight) into the root-of-trust, we only migrate the control policy and its state transition logic into a data capsule, which is a small portion of the controller's code (195 lines of code). Access to the data capsule which is placed in isolated memory protected by the micro-hypervisor, is then mediated by the micro-hypervisor via code white-listing [58, 93]. This scheme ensures that the data capsule can only be accessed by the middlebox management application. The original code base is then refactored to access these data capsules (see details in 7).

5.3 vTPM for Component Instance Validation

The code of critical software components (e.g., middleboxes) must be protected from malicious modifications. Instead of carving out individual pieces of these components, we relax our protection to only identifying changes to the code using remote attestation [7, 16] via a *vTPM*. This limits the TCB increase to only the attestation operations and measurements. This combination of isolation and attestation allows us to maintain a small TCB with good per-packet performance.

Our Approach: Attestation is often provided by a Trusted Platform Modules (TPM). As a physical TPM is not available on all hardware, we leverage a software implementation, a *virtual trusted platform module (vTPM)* in the micro-hypervisor. We utilize a subset of its capabilities to securely store a chain of measurements, by extending a platform configuration register (PCR), and its ability to securely provide the stored values (i.e., a PCR quote). These vTPM measurements can be trusted as access to the vTPM is mediated by the micro-hypervisor and the PCR values are placed in a data capsule, precluding an attacker from maliciously altering the PCR values. While the vTPM does not provide persistent secure storage and attestation keys by itself, it has performance advantages compared to a physical TPM (e.g., not limited by physical memory or data bus).[3] This allows applying vTPM measurements at a fine granularity, providing increased PCRs (100+) and lower access latency (0.9 msecs).[4]

5.4 Packet Signing for Packet Path and Data Validation

To provide packet path and data validation on the gateway, a strawman solution is to use existing secure tunneling protocols (e.g., IPsec, TLS). Unfortunately, this has two limitations. (1) Most tunneling implementations rely on the OS to protect secret keys files (e.g., .ssh directory), allowing an attacker who has compromised the OS to craft secure messages. (2) Exiting tunneling approaches are too heavyweight for traffic on the gateway (data path traffic). As we show in §9.3, when

[2] A typical SDN controller (e.g., NOX, ONOS, OpenDayLight) has a code base from 20-300k lines of code, which is an order of magnitude larger than many micro-hypervisors.

[3] Note that a vTPM can be bridged with a platform hardware (physical) TPM, if available, to provide persistent secure storage and attestation keys.

[4] Comparison between physical TPM and vTPM in Appendix A.

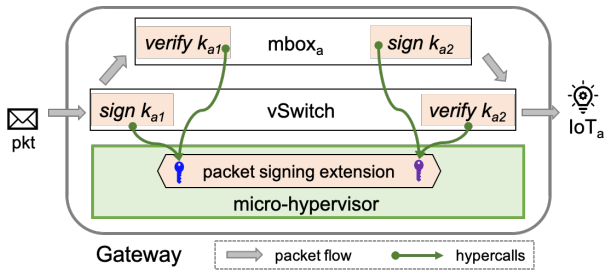


Fig. 5: Packet signing operations to verify packet path and data between the vSwitch and middleboxes.

enforcing tunneling between the vSwitch and middleboxes for every packet, it could reduce packet processing throughput by greater than 67% of the baseline throughput.

Our Approach: Inspired by prior work on path verification protocols (e.g., [39, 48]), we design a simple *packet signing* mechanism to enforce each packet follows the correct path with the correct data. Figure 5 shows our packet signing mechanism. We extend the vSwitch and each middlebox by adding two functions: *sign* and *verify* (see details in §7). Both functions use keys stored in the micro-hypervisor to generate a digital signature. The *sign* is triggered when sending a packet and *verify* is called when receiving a packet.

For example, a packet *pkt* arrives at the vSwitch, which looks up its intended path to $mbox_a$ (configured by the controller). Then vSwitch calls *sign* to create a digital signature over the entire packet (header and payload) using key k_{a1}, a unique key shared between the vSwitch and the $mbox_a$. After the middlebox receives the packet, it uses its key k_{a1} to *verify* the signature. If the packet has been tampered with or routed to the wrong destination (e.g., Attack 3 in §2.2), the verification fails and the packet is dropped. After the middlebox processes the packet, it *signs* the packet using another key (e.g., k_{i2}). Note that this is necessary as the middlebox might modify the packet data, resulting in the previous signature being *obsolete*. The vSwitch similarly verifies this signature.

Compared to traditional tunneling, our verification approach is more lightweight. It does not require expensive setup (unlike TLS) and uses a simpler verification header (unlike IPSec). Furthermore, the micro-hypervisor provides assurances that the digital signatures can be trusted, as it protects the secret keys and sign/verification functions, thereby stopping an attacker from forging signed packets. Thus, our signing mechanism ensures that packets follow the correct paths at a low performance overhead (see comparison in §9). Next, we discuss how we secure the control channel.

5.5 Trusted Agent for Control Message Integrity and Authentication

Existing SDN architectures often support an encrypted message exchange mechanism (e.g., IPsec/TLS) for the control channel. Unfortunately, these do not protect the secret keys.

Our Approach: We add trusted agents on the controller and gateway to secure control channel traffic. These agents intercept all control channel messages and protect the secret keys in a data capsule. Further, micro-hypervisor mediation ensures only trusted applications can access the secret keys. We assume that the secret keys are exchanged out of band and that a unique pair exists for each controller and gateway set. Since the secret keys and agents are memory-protected and isolated by the micro-hypervisor, they are immune to attacks from untrusted components including the OS.

Table 3: Summary of design components for achieving trust requirement in order to mitigate attacks from §2.2.

Attack in §2.2	Requirement	Defense
Attack 1	TR_{sw1}	Limit access via data capsule
Attack 2	TR_{sw2}	Attest component via vTPM
Attack 3	TR_{comm1}	Identify modification via packet signatures
Attack 4	TR_{comm2}	Protect secret keys via trusted agent

5.6 Design Summary

Table 3 summarizes key components of our system design, the trust requirements they address and the class of attacks they protect against. Our system design relies on a micro-hypervisor root-of-trust to provide the needed trust capabilities at a low-cost, running legacy software on a broad base of existing hardware. Our performant, low-TCB hypervisor extensions (data capsule, vTPM, packet signing, and trusted agent) work in synergy towards achieving our overarching trust property. As shown in Table 3, the data capsule (TR_{sw1}) blocks an attacker from modifying the security policy (Attack 1). Attestation via the vTPM (TR_{sw2}) detects an attacker modifying a middlebox (Attack 2). Packet signatures (TR_{comm1}) mitigates local attackers modifying packets (Attack 3). Finally, trusted agents (TR_{comm2}) blocks an attacker from forging control channel messages (Attack 4).

6 Lightweight Middleboxes for Scalability

In the previous sections, we have described how we can design a trustworthy gateway based on a low-cost root-of-trust. Another key challenge is scalability, where a single gateway can protect all IoT devices in a deployment (e.g., a home environment). In this section, we first identify the scalability bottlenecks and then present two of our optimizations.

6.1 Identifying Bottlenecks

We start with testing how many middleboxes a low-cost Raspberry Pi 3 can run simultaneously. Since each middlebox is assigned to one IoT device, this test shows the maximum number of IoT devices the current platform can protect. We pick Snort [14] as an example middlebox because an intrusion prevention system (IPS) is likely required by all deployments. For each Snort instance, we run it using the default configuration with the full community rule set [14] to provide broad coverage. Unfortunately, we could only run three snort instances simultaneously, which cannot even support the current average US household (∼8 devices [13]), not to mention future growth.

When running multiple Snort instances, we noted that the main bottleneck limiting scalability was the memory required by each Snort instance. One Snort instance consumes 452

MB of memory. We used Intel's VTune Amplifier profiling tool to identify the most significant contributors to memory consumption and found that the majority of the memory was allocated on the heap for rules and their processing, followed by socket buffers. Based on these findings, we propose two optimizations to reduce the memory consumption:

6.2 Optimizations

Customized Configurations: Our first optimization is to customize Snort configurations based on the protected IoT device. Our analysis shows that the memory required by a Snort instance is directly proportional to the number of rules it is configured with (∼27.5 KB per rule). The full community rule set, composed of 10,918 rules, is designed to protect a wide array of devices. Our idea is to use custom profiles that only contain the rules that are applicable to that particular device (e.g., an IoT device running Linux does not need Windows rules). Thus, we categorize the community rules (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. This customization realized up to a five-fold decrease in Snort rules.

We also optimize the socket buffer, the second largest memory consumer. Snort has a default socket buffer of 166 MB, to support analyzing network traffic from multiple devices at rates of 200-500 Mbps. For our use case, supporting such high throughput is unnecessary, as each IoT device has its own instance of Snort and peak bandwidths of less than 20 Mbps[5]. Using this observation, we reduce the socket buffer and free up 163 MB per Snort instance.

Sharing Rules: After categorizing rules for each device, we noted that many of the rules were still common across multiple IoT devices, with only a small fraction of the rules being device-specific. This results in the same rule being in memory multiple times (i.e., once for each IoT device being protected). Our insight is to place these common rules into a shared memory region, so that we only have one instance of the rules in memory (similar to [53]). Subsequent instances can be instantiated at a significantly reduced memory footprint (e.g., 30 MB per instance). Combining these optimizations, reduces the memory footprint of Snort by more than 12x per instance, allowing a single hardware platform to support more than 50 simultaneous Snort instances (Fig. 7a).

Our approach and optimizations are general and can apply to other platforms and middleboxes as well. For instance, many other low-cost platforms (e.g., OpenWRT routers) have less than 2 GB of RAM giving them the same bottleneck. Similarly, other middleboxes (e.g., Zeek [107], Suricata [81]) that use a common set of rules across multiple devices, could also benefit from our analysis approach and optimizations. Next, we discuss a prototype implementation of JETFIRE.

[5] In sampling 10 commodity home IoT devices, we noted a peak throughput of less than 1 Mbps. For additional details, see Appendix A.

7 Implementation

We implemented a JETFIRE prototype using two Raspberry Pi 3Bs, one for the controller and the other for the gateway. Both use the uberXMHF micro-hypervisor framework [89] and Raspbian Jessie (Linux 4.4.y). For the controller, we use OpenDayLight (Aluminum), the largest open source SDN controller. For the gateway, we run Dockerized middleboxes (e.g., Snort, Squid, iptables) with OpenvSwitch (OVS 2.12.1) for packet routing. Next, we describe uberXMHF and how we extend it to achieve our trust properties.

uberXMF micro-hypervisor: Our trust architecture is built on top of uberXMHF, an open-source,[6] formally verified, micro-hypervisor framework [89, 90, 91]. We chose uberXMHF because it supports both x86 and ARM platforms and provides a modular framework for compositional verification. This allows for adding hypervisor extensions while preserving the core micro-hypervisor's memory integrity without needing to repeat the verification [89].

We use uberXMHF (v5.0) and realize our protections (§5) as hypervisor extensions. Each hypervisor extension exposes a hypercall interface that is callable from both user and kernel space and allows for transferring up to 4 KB of data. Our *security policy* extension stores and transitions each IoT device's current FSM state, it prohibits adding states by setting the maximum number of states for each device's FSM and limiting this to only occur once. Our *packet signing* extension uses the micro-hypervisor's internal cryto library to perform an SHA256-HMAC on an input data buffer using secret keys stored in the micro-hypervisor. Our *trusted agent* uses a secret key in the hypervisor to encrypt a data buffer using AES. The lines of code for each hypervisor extension is given in Table 5.

Trusted Data and Code (TR_{sw1}, TR_{sw2}): We integrate our security policy into our modified OpenDayLight controller so that each policy query goes to the micro-hypervisor (to achieve TR_{sw1}). As the hypervisor extensions are in C and the controller in Java, we build a shared library that performs the hypercalls and leverage a Java Native Interface to integrate these into the controller's operations. For attestation, we create a Python daemon that measures each middlebox's executables and configuration. Measurements are stored using the micro-hypervisor's PCR extend interface, and then sent to the controller (as a vTPM quote), and checked against the value in the security policy (to achieve TR_{sw2}). These measurements and quotes are then repeated periodically (e.g., 1 min).

Trusted Communications (TR_{comm1}, TR_{comm2}): We integrate our packet signing into OVS and our Dockerized middleboxes (to achieve TR_{comm1}). Within OVS, we add two *new actions* (sign and verify) to both the user and kernel space virtual switch functionality (where the kernel module realizes a 2x throughput increase (§9.3). The sign function appends the signature returned by the hypercall to the packet's payload. For this added data to arrive at the middlebox, the packet's

[6] https://uberxmhf.org

headers are modified to account for the increased packet length. The verify function strips the signature from the packet, recalculates the packet header, and performs a hypercall to verify whether the two signatures match.

For middleboxes, we leverage `NFQUEUE` interfaces to implement our signing. Since `NFQUEUE` can intercept both received and output packets, we added a userspace callback to perform operations similar to the sign and verify actions added to OVS. This allows for an unmodified packet to be analyzed by the network function (e.g., Snort). We also integrate our trusted agent into the middleboxes (to demonstrate TR_{comm2}). We implement a Python daemon that checks the middlebox's log files for alerts. Upon a modification, the daemon performs a hypercall to encrypt the new data prior to sending it to the controller which decrypts the data using a hypercall to its trusted agent (integrated similar to the security policy).

SDN Controller and Example Middleboxes: Realizing our prototype system required extending the OpenDayLight controller (adding approximately 2k lines of code). This includes adding functionality for remotely configuring the Dockerized middleboxes (leveraging the Docker API), sending flow rules to that included our added actions (as these are not a part of the OpenFlow protocol), and integrating remote attestation of middleboxes. Additionally, we realize example middleboxes running commodity network functions: (1) Snort IPS to block known vulnerabilities, (2) `iptables` as a firewall, and (3) Squid authenticating HTTP proxy to add authentication for devices with default credentials.

8 Security Evaluation

We analyze the security of JETFIRE's design along four axis: (1) model-based validation of our design §8.1, (2) the architecture's robustness to attacks from the SDN literature §8.2, (3) validation of our implementation using synthetic attacks §8.3, and (4) measure the increase of the micro-hypervisor's TCB §8.4.

8.1 Validating Our Design

We build a formal model of our software-defined security gateway to specify our trust properties (§4). We describe this model and then evaluate it using bounded model checking to validate our design is secure by construction.

8.1.1 Formal Model Description

We specify our software-defined gateway model using the Alloy modeling language[28]. We briefly introduce Alloy and before describing our model.

Alloy Modeling Language: Alloy models are defined using first-order, relational logic. At its core, the Alloy language is an easy to use but expressive logic based on the notion of relations, and was inspired by the Z specification language and Tarski's relational calculus [28]. The Alloy model is compiled into a scope-bounded satisfiability problem and analyzed by off-the-shelf SAT solvers. We use this analysis to identify *counter examples to constraints* and verify our trust properties.

Listing 1 Abridged formal model of JETFIRE's trusted software-defined security gateway architecture.

```
1:  sig Controller {
2:      policy : one Policy,
3:      apps: set Application,
4:      controlchannel : one Channel
5:  }
6:  sig Gateway {
7:      vswitch : one vSwitch,
8:      mbox : set Middlebox,
9:      controlchannel : one Channel
10: }
11: function ProcessPkt(pkt : Packet, g : Gateway)
12:     g.mbox_i = RoutePkt(pkt, g.vswitch)
13:     pkt.state = MiddleboxProcess(pkt, g.mbox_i)
14:     if pkt.state == Benign then
15:         pkt.action = Allow
16:     else
17:         pkt.action = Drop
18:     return pkt.action
19: pred TrustedGateway(g : Gateway, c : Controller)
20:     TamperProof(c.policy)
21:     SecureChannel(g.controlchannel, c.controlchannel)
22:     RemoteAttest(c.apps)
23:     RemoteAttest(g.vswitch)
24:     for g.mbox_i in c.policy do
25:         RemoteAttest(g.mbox_i)
26:         AuthenticateRoute(g.vswitch, g.mbox_i)
27: assert ProcessPktCorrectly(g : Gateway, pkt : Packet)
28:     TrustedGateway(g)
29:     pkt ∈ BenignPkts ⟹ ProcessPkt(pkt, g) == Allow
30:     pkt ∈ MaliciousPkts ⟹ ProcessPkt(pkt, g) == Drop
```

Software-defined Gateway Model: JETFIRE's software-defined IoT security gateway model consists of a centralized controller and a set of gateways that process packets to and from IoT devices. For brevity, we discuss an example architecture with a single gateway to explain our abridged Alloy model in Listing 1 (the full model can be found in [15]).

1. Controller and Gateway. We first model two key entities: a *Controller* and a *Gateway* using Alloy's `sig` interface (lines 1-10). A `sig`, or signature, defines a set (i.e., Controller) and its relationship to other sets (i.e., each Controller has one Policy, line 2). The controller maintains the security policy, and the control applications use the control channel to configure each gateway based on the policy. Each gateway runs one vSwitch and a set of middleboxes. The gateway receives commands over the control channel for instantiating middleboxes and installing paths in the vSwitch. Each path specifies which middlebox a specific IoT device's traffic should be routed through.

2. ProcessPkt. We model how the gateway processes packets using Alloy's `function` interface (lines 11-18). A `function` evaluates a series of statements and returns all possible solutions. A packet received by the gateway is sent to the vSwitch for routing. The vSwitch routes the packet to the specific middlebox (line 12). Then the middlebox processes the packet and determines if the packet is benign or malicious (lines 13). Benign packets are routed back to the switch and sent to the IoT device while all other packets are dropped.

Table 4: JETFIRE's mitigation of known SDN attacks.

Attack Type	Example Attack	Our Defense	Mitigates
(a) Controller Application	A1: Manipulate controller application's state [103]	Data Capsule + vTPM	✓
	A2: Manipulate controller application's operations [37, 95, 103]		✓
	A3: Manipulate command or variable [37, 76]	Data Capsule	✓
(b) Control Channel	A4: Sniff messages [37]	Trusted Agent	✓
	A5: Inject messages [18, 37, 95, 100, 103]		✓
	A6: Modify messages [37, 95, 103]		✓
(c) Gateway Application	A7: Subvert middlebox execution [56, 86]	Data Capsule + vTPM	✓
	A8: Manipulate command or variable [37, 76]	Data Capsule	✓
(d) Data Channel	A9: Modify packet path [45]	Packet Signing	✓
	A10: Modify packet data [45]		✓

3. TrustedGateway. Next, we define a trusted gateway (Eq 2) using Alloy's `pred` interface (lines 19-26). A `pred`, or predicate, evaluates a series of constraints. It returns true only if all the constraints are met and false otherwise. Thus, the following conditions must all be met for a gateway to be trusted. First, an attacker must not be able to tamper with the policy on the controller (line 20, Eq 3) Second, the control channel between the controller and the gateway must be secure so that it is immune to an attacker injecting malicious messages (line 21, Eq 6). Third, the correct software must be running on the controller, vSwitch, and middlebox (lines 22-25, Eq 4). Finally, each packet must follow the path specified by the controller. This must be enforced by the vSwitch and each middlebox (line 26, Eq 5). If all of these conditions are true then the gateway is trusted.

4. ProcessPktCorrectly. Finally, we define our goal (Eq 1) that all output packets were processed correctly using Alloy's `assert` interface (lines 27-30). In Alloy, an `assert` claims that a series of statements must be true based upon the model, and will generate a counter example if any of the claims do not hold to be true. A trusted gateway achieves the goal of allowing all benign packets while dropping all malicious packets.

8.1.2 Model Evaluation

We analyzed our system model up to a bound of 100 (i.e., 100 instances of each `sig`) and were unable to identify a counter example resulting in the model outputing a packet processed by an incorrect middlebox.[7] Additionally, we systematically removed constraints related to our trust requirements (e.g., middlebox code does not need to be attested, violating TR_{sw2}) and verified that each resulted in a counter example that violated our overarching trust property. This analysis provides confidence in our our system's design and trust requirements.

Our Alloy model aided in identifying nuances and helped us refine our design. The model highlighted the need to prohibit packets from completely skipping a middlebox. For example, if a middlebox signs input and output packets with the same key it allows a packet to bypass the middlebox without being detected. Similarly, our model highlighted software components that either needed to be trusted or be regularly attested in order to trust the system's operation (e.g., the controller software). Next, we extend this model to evaluate JETFIRE's applicability beyond this use case, and look at securing broader SDN architectures.

[7] This analysis can be performed on a personal computer (see Appendix A).

Listing 2 Example Alloy analysis of prior SDN attack, search for ability of attacker to modify a controller application's state.

```
1: pred CANMODAPPSTATE (c : Controller, a : Apps) {
2:     a in c.apps ∧ a.state == Exposed
   }
3: assert ATTACKERNOTMODTRUSTAPPSTATE(c : Controller, a : Apps)
4:     TRUSTCONTROLLER(c)
5:     a.state ==DataCapsule ⟹ CANMODAPPSTATE(c, a) == False
```

8.2 Robustness to Prior SDN Attacks

We further evaluated JETFIRE's system model against 10 representant SDN attacks from the literature (summarized in Table 4) [18, 37, 45, 95, 100, 103]. To identify if our system could protect against these attacks, we extend our Alloy model. Listing 2 provides an example extension for checking if a controller application's software state can be modified (the full model of all attacks can be found in [15]). This example verifies an attacker cannot modify an application's state if it is protected by a data capsule. The attacks we analyze fall into the following four groupings based upon the attack's target: (a) controller applications, (b) control channel, (c) gateway applications, and (d) data channel. We discuss each of these attack types below.

(a) Controller Application Attacks: An attacker compromising the controller or a controller application could alter its state or operations (e.g., controller's global network view [103]). JETFIRE can defend against this type of attack by placing the critical data (e.g., security policy §5.2) into a data capsule, and use a vTPM to attest other pieces.

(b) Control Channel Attacks: Attacks could tamper with an established control channel between the controller and gateway by injecting malicious flow rules into the vSwitch [103]. Our trusted agent (§5.5) on the controller and each gateway can prevent this type of attack by using encrypted and authenticated channel. Further, JETFIRE can mitigate an attacker on the gateway/controller from accessing the secret keys and sending malicious messages from a compromised host.

(c) Gateway Application Attacks: Attackers could attack applications running on the gateway including middleboxes and the vSwitch. For example, an attack could change the vSwitch's routing rules or modify a middlebox's binary [56]. These will

result in incorrectly processing network traffic (e.g., disableing a firewall's drop action). As discussed in §5.3, a combination of vTPM attestation and data capsule isolation can be used to detect such attacks.

(d) Data Channel Attacks: Attackers that have tampered with the OS can modify a packet's processing path or its data, such as bypassing a middlebox or modifying a packet payload. These could result in incorrect gateway operations (e.g., allow a malicious packet the firewall should have dropped). JETFIRE's per-packet signing mechanism (§5.4) can detect such data channel modifications.

This analysis implies our architecture's applicability beyond software-defined IoT security gateways and could be used for securing a wider array of SDN-based architectures. Future extensions such as confidential storage, controller DoS protections, and topology verification could provide additional guarantees against prior attacks.

8.3 Synthetic Attacks on the Prototype

Beyond analyzing our system model, we generated synthetic attacks to validate that our prototype implementation provided each of our trust requirements (§4). We discuss each below.

(a) Rogue security policy modification (testing TR_{sw1})**:** We simulate an attacker with local access to the controller attempting to modify the controller's security policy (i.e., loading new values). We verified that the micro-hypervisor access mediation (via code white listing) denies this process access to the security policy (as only the security policy application has access), and that the security policy remains unchanged.

(b) Booting a modified middlebox image (testing TR_{sw2})**:** We start a modified middlebox to simulate an attacker tampering with a middlebox. The controller detects this misconfiguration (based upon the PCR quote it received) within 10 seconds of the middlebox booting.

(c) Malicious control channel injection (testing TR_{comm1})**:** We inject false middlebox alert messages over the control channel to simulate an attacker attempting to change the middlebox on the gateway. As these messages did not go through the trusted agent, the messages were dropped by the controller for failing authentication.

(d) Send packets on wrong path (testing TR_{comm2})**:** To simulate an attacker sending packets to the wrong middlebox, we sign packets with the wrong key (to generate an invalid signature). These malicious packets were injected both before and after the middlebox process the packet to demonstrate both OVS and the middlebox drop these invalid packets.

These validation tests gave us confidence that our implementation achieves our trust requirements. A more robust guarantee about our implementation could be achieved using code verification, we leave this to future work.

8.4 TCB of Micro-hypervisor and Extensions

Recall that one of our challenges §3.4 is to achieve our trust properties while keeping a small TCB. Our baseline is

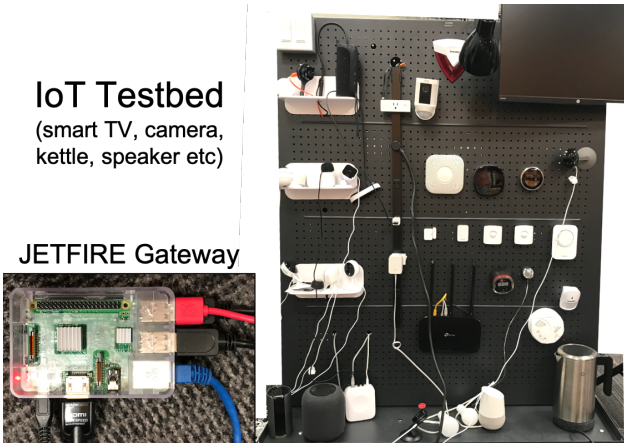


Fig. 6: Smart home IoT testbed deployment being protected by our prototype JETFIRE security gateway.

the uberXMHF micro-hypervisor used in our implementation, which itself has a small TCB (5544 source lines [90]) and has been formally verified [88, 91]. As shown in Table 5, we add three main hypervisor extensions, each extensions was implemented in less than 200 source lines of code. All of them add a total increase of 6.6% of TCB size. This keeps the micro-hypervisor code base amenable to (future) formal verification (§10) as demonstrated by uberXMHF's x86 verification [91].

Table 5: The impact of Jetfire's extensions on TCB size.

Hypervisor Extensions	Lines of code	Percent increase
Data capsule	195	3.5%
Packet signing	70	1.3%
Trusted agent	102	1.8%
All extensions	367	6.6%

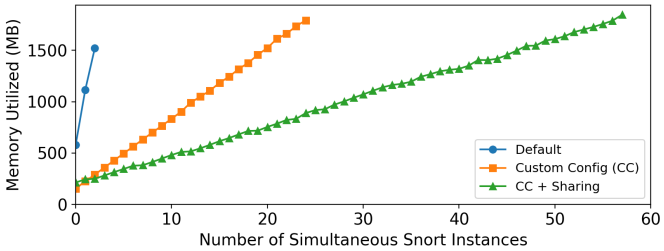
9 Performance Evaluation

Using our proof of concept implementation, we deployed the system in a simulated IoT deployment(§9.1). This demonstrates the ability to retrofit security into the deployment and protect devices against various known network attacks.

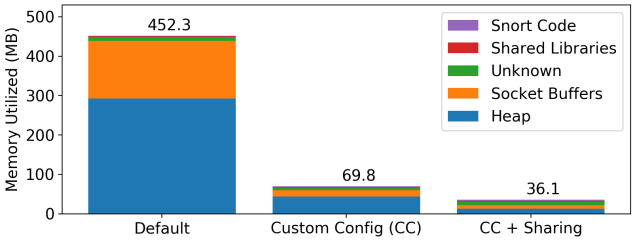
9.1 Case Deployment

To demonstrate JETFIRE ease of deployment, we applied it to protect a simulated smart home environment. These showcase the diversity of security vulnerabilities it can patch without impacting operations.

Smart Home IoT Testbed Deployment: Our simulated home IoT deployment contained 17 IP-based, commercial home IoT devices including: smart cameras, smart doorbells, smart plugs, and smart lights (testbed shown in Fig. 6). These devices connected to the JETFIRE gateway, which served as their WiFi access point and deployed device-specific middleboxes for each connected device. We then demonstrated the gateway's ability to patch two known vulnerabilities. (1) For camera with default credentials, its traffic was routed through a Squid authenticating HTTP proxy to add an extra layer of authentication. (2) A network attached storage (NAS) with an unpatched SambaCry vulnerability had its traffic routed through a Snort IPS



(a) Simultaneous Snort instances realized for each optimization.



(b) Virtual memory profiling of single Snort instance.

Fig. 7: Scalability evaluation of the number of simultaneous Snort instances on a Raspberry Pi 3B after optimizations.

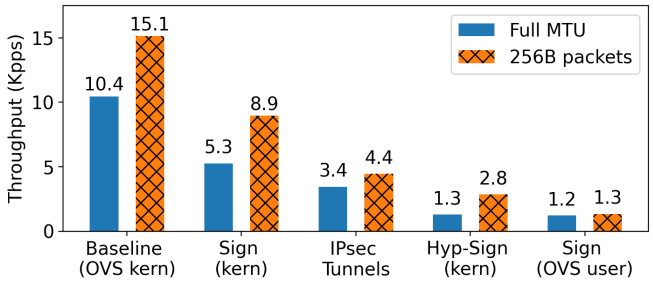


Fig. 8: Packet signing impact on median packets per second (pps) throughput for user and kernel vSwitch.

that detected and blocked this exploit. Next, we measured the latency latency increases for users' HTTP requests when going through the gateway. A small latency increase was noted, with a median increase of 11.5 milliseconds. The increased latency is much lower than the threshold (110 milliseconds) where users error rates are noted to increase [54], which suggests that JETFIRE can be deployed to provide trusted security for smart homes without impacting users' experience.

9.2 Scalability

For our architecture to be deployable in many settings (e.g., smart homes, factories), a single hardware gateway needs to support small deployments (<20 IoT devices, §3.1). This scalability is highly dependent upon the middlebox being used. We utilized the most widely deployed IPS, Snort [14], as we anticipate each IoT device requiring this security functionality.

As shown in Figure 7a, applying the optimizations discussed in §6, we achieved a 19x increase in the number of simultaneous Snort instances. In particular, custom configurations (CC) enabled an 8x increase (24 instances, average 69.8 MB/instance), and utilizing both custom configurations and shared memory enabled an additional 2.4x increase (57 instances, average 36.1 MB/instance).

We noted a minimal impact on per-packet latency (with sharing reducing latency) that these scalability optimizations had on HTTP GET requests. As anticipated, the reduced configurations had similar latency. Moving the signature rules into shared memory reduced the median latency by 4.3 milliseconds (52% reduction). We hypothesize that this latency reduction is from the shared memory not being evicted from the cache during context switches.

9.3 Packet Processing Throughput

To protect data packets, Jetfire uses a lightweight signing mechanism to add signatures for each packet (§5.4) in vSwitch. In this experiment, we evaluate its throughput impact. As shown Fig. 8, our baseline is 'OVS kern', which runs original OVS kernel module for routing without involving any extra overhead. First, we compare our kernel signing ('Sign-kern') with IPsec tunnels, both without protection. We noted our packet signing provided an additional 18% throughput for full MTU, and 19% for 256 Byte packets. Second, we compare our hypervisor *protected* packet signing ('Hyp-Sign') with an alternative approach that performs signing in an enclave. As noted earlier, trusted enclaves can only support user space applications, thus attempting to enable signing for vSwitch in an enclave would require it operates in user space. We use OVS packet signing in user space ('OVS user') to emulate this approach. While this comparison favors our approximation of a secure enclave as the real enclave implementation would add extra overhead (e.g., memory copying), our approach still outperforms this situation, particularly for smaller sized packets.

To further understand the impacts from the micro-hypervisor protected signing, we microbenchmarked this operation. We noted that the packet signing overhead is ∼1 millisecond (544 μseconds for HMAC and 519 μseconds for hypervisor call).

Impact on Real Deployment: In order to determine the throughput impact in the real IoT deployment, we sampled a number of commodity IoT devices during normal user interaction. We measured their median steady-state throughput to approximate network utilization as well as the average packet size (see details in Appendix A). We noted that many devices have low steady-state throughput (<80 pps), with a small average packet size (<325 bytes). Additionally, the traffic is very bursty, where the majority of devices had more than 30 seconds between bursts. This burstiness makes our 1 millisecond latency increase only impact a small percentage of IoT devices. Thus, with many IoT devices having low network throughput and send in bursts, a security gateway with a throughput of 1.6 Kpps could likely support 20 devices without creating a significant impact on device operations.

10 Discussion

Use Cases Leveraging a Trusted Architecture: A JETFIRE architecture could serve as the foundation for advanced ca-

pabilities that require increased trust in their operation. For example, other work has looked at granting middleboxes the ability to decrypt TLS data to enable functions such as IPS on end-to-end encrypted data [49, 98]. These encryption keys could be protected by the micro-hypervisor, and all processing on the cleartext data be performed in the hypervisor. Providing a guarantee that the encrypted data remains protected from an attacker while enabling the detection of malicious packets in an encrypted stream. Similarly, this trusted foundation could be used for performing active deception, ensuring that an attacker cannot control the deception to hide their actions.

Strengthening Trust Properties: Our trust property can be further strengthened (e.g., any packet sent to the gateway is processed by the correct middlebox) by incorporating protections with the NIC driver. We identify two potential approaches: (1) adding a layer of sign/verify at the NIC kernel driver, and (2) placing the NIC driver as an extension within the micro-hypervisor. This will enable JETFIRE to mitigate DoS attacks where the attacker maliciously drops packets on the gateway.

Increasing Hardware Capabilities: IoT devices currently have low network utilization, many sending <3 Mbps. In the future, their network utilization is likely to increase; however, more capable hardware platforms will also be available at low-cost. For example, our prototype platform (Raspberry Pi 3B) was limited by its network hardware capabilities (limited by the USB 2.0 bus). The Raspberry Pi 4 has removed this bottleneck, providing a 1 Gbps NIC (in addition to increased CPU and memory). These hardware changes make the CPU the bottleneck, as the network throughput of JETFIRE on a Raspberry Pi 4 is dependent upon the complexity of the operation being performed on the network traffic (with throughput differences of >200 Mbps between a Snort IPS and an `iptables` firewall).

Future Work: This paper demonstrates that a micro-hypervisor based system architecture can be used for creating an efficacious trusted software-defined security gateway. We demonstrate that the design is secure (§8). Future work involves formally verifying the micro-hypervisor extensions to ensure their implementation does not create vulnerabilities.

11 Related Work

We group related efforts into two categories: trusted computing and SDN security. Our architecture leverages trusted computing works to build a practical solution to defend a wide range of attacks uncovered by prior SDN security efforts.

Trusted Computing: We leverage prior works on trusted computing to create a practical architecture for trusting software-defined IoT security architectures (e.g., which have often placed the entire security architecture within the TCB [35, 105, 106]). Hypervisors have been used to provide security primitives such as isolation, mediation, and attestation [41, 46, 62, 84, 89, 93]. Micro-hypervisors have been shown to support a variety of hardware platforms (x86 [77, 88], ARM [90], microcontroller [4]) running unmodified software (e.g., Linux)[4, 85, 89]. A primary use of TPMs is providing remote attestation [7], leading to multiple software implementations [64, 89]. Secure routing proposals have used signatures to verify packet paths [39, 48].

SDN Security: Prior works adding trust to SDN architectures have only analyzed securing pieces. Piecemeal approaches have been employed for adding trust by (1) using trusted hardware (e.g., SGX) on data planes [56, 86], (2) adding security features to SDN controllers [57, 76], and (3) developing tools for analyzing consistency between the control and data plane.

For the data plane, trusted hardware (e.g., SGX [56, 86], MPX [111]) has been looked to for providing increased security guarantees about middleboxes running in untrusted cloud environments. These approaches provide code attestation, confidentiality, and mediation[8] [56], and memory access boundaries checking [111]. Unfortunately, they are not ideal for IoT as they have high performance overheads and lack generality (limited to a specific CPU and only supporting user space applications with constrained memory allocations).

For the control plane, researchers have focused on mediating multiple applications on the controller by adding permissions [30, 72, 76, 97]. Others have looked to ensure consistency of routing rules generated by separate applications [29, 57]. Our work looks to support these controllers and provide the ability to provide foundation for guaranteeing trust in their operations.

Others have developed tools to ensure consistency between the control and data planes with respect to packet routing, creating tools for identifying forwarding anomalies [2, 12, 20, 25, 31, 32, 33, 60, 80, 82, 109, 110] and SDN-specific attacks [18, 37, 66, 99]. Unfortunately, none of these provide runtime protections against our threat model.

12 Conclusions

This paper addresses a fundamental question for future IoT deployments: *Can we create a foundation for trustworthy gateway architectures to retrofit security onto IoT deployments with potentially insecure devices?* In designing JETFIRE, we tackled key challenges in providing practical foundations for trust and ensuring scalable yet low-cost capabilities for fine-grained security postures. JETFIRE is trustworthy by construction and is backed by a formal validation of its design and interfaces. Our evaluation shows that JETFIRE can serve as the basis for a low-cost, deployable, and trustworthy foundation for future IoT security gateways. Using JETFIRE, we can support IoT deployments with 50+ IoT where each has a customized IPS (Snort) module running in a single Raspberry Pi 3B gateway. We plan to open source our formal models and end-to-end implementation [15] to spur further innovations for different IoT deployments and allow others to build on our work.

[8] Mediation requires re-implementing the middlebox software.

References

[1] Researcher: Backdoor mechanism still active in many iot products. https://www.zdnet.com/article/researcher-backdoor-mechanism-still-active-in-many-iot-products/, 2020.

[2] Ehab Al-Shaer et al. FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures. In *SafeConfig*, 2010.

[3] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE S&P*, 2019.

[4] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. Sμv - the security microvisor: A formally-verified software-based security architecture for the internet of things. *IEEE Trans. Dependable Sec. Comput.*, 16(5):885–901, 2019.

[5] Manos Antonakakis et al. Understanding the mirai botnet. In *USENIX Security 17*, Vancouver, BC, 2017. USENIX Association.

[6] ARM. Trustzone - arm developer. https://developer.arm.com/ip-products/security-ip/trustzone, 2020.

[7] Will Arthur, David Challener, and Kenneth Goldman. *History of the TPM*. Apress, Berkeley, CA, 2015.

[8] David Barrera, Ian Molloy, and Heqing Huang. Standardizing iot network security policy enforcement. In *DISS 2018*, 2018.

[9] Bit defender box 2. https://www.bitdefender.com/box/, 2018.

[10] A. Cenedese, A. Zanella, L. Vangelista, and M. Zorzi. Padova smart city: An urban internet of things experimentation. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6, 2014.

[11] Chen Chen, Petros Maniatis, Adrian Perrig, Amit Vasudevan, and Vyas Sekar. Towards verifiable resource accounting for outsourced computation. In *ACM VEE*, 2013.

[12] Po-Wen Chi, Chien-Ting Kuo, Jing-Wei Guo, and Chin-Laung Lei. How to Detect a Compromised SDN Switch. In *NetSoft*. IEEE, 2015.

[13] Cisco. Cisco visual network index. https://www.cisco.com/c/m/en_us/solutions/service-provider/vni-forecast-highlights.html#, 2019.

[14] Cisco. Snort. https://www.snort.org, 2019.

[15] Jetfire code. http://www.filedropper.com/usenixtar, 2020.

[16] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10(2), 2011.

[17] Cujo. https://www.getcujo.com, 2018. Accessed: 2018-03-23.

[18] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. Sphinx: Detecting security attacks in software-defined networks. In *Ndss*, volume 15, pages 8–11, 2015.

[19] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proc. of CCS*, 2008.

[20] Mihai Dobrescu and Katerina Argyraki. Software dataplane verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 101–114, 2014.

[21] Naganand Doraswamy and Dan Harkins. *IPSec: the new security standard for the Internet, intranets, and virtual private networks*. Prentice Hall Professional, 2003.

[22] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *Proc. of IEEE/ACM ASE 2010*, 2010.

[23] Branden Ghena, William Beyer, Allen Hillaker, Jonathan Pevarnek, and J. Alex Halderman. Green lights forever: Analyzing the security of traffic infrastructure. In *Proceedings of the 8th USENIX Conference on Offensive Technologies*, WOOT'14, page 7, USA, 2014. USENIX Association.

[24] Stephen Herwig, Christina Garman, and Dave Levin. Achieving keyless cdns with conclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 735–751. USENIX Association, August 2020.

[25] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 735–749, 2017.

[26] IHS. Internet of things (iot) connected devices installed base worldwide from 2015 to 2025 (in billions). www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, 2018.

[27] Intel. Intel software guard extensions: Developer guide. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf, 2016.

[28] Daniel Jackson. Alloy: A new technology for software modelling. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 20–20, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[29] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 87–101, 2015.

[30] Heedo Kang, Seungwon Shin, Vinod Yegneswaran, Shalini Ghosh, and Phillip Porras. Aegis: An automated permission generation and verification system for sdns. In *Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges*, pages 20–26, 2018.

[31] Peyman Kazemian et al. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*. USENIX, 2013.

[32] Ahmed Khurshid et al. Veriflow: Verifying network-wide Invariants in Real Time. In *NSDI*. USENIX, 2013.

[33] Hyojoon Kim et al. Kinetic: Verifiable Dynamic Network Control. In *NSDI*. USENIX, 2015.

[34] Tiffany Hyun-Jin Kim, Cristina Basescu, Limin Jia, Soo Bum Lee, Yih-Chun Hu, and Adrian Perrig. Lightweight source authentication and path validation. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[35] Ronny Ko and James Mickens. Deadbolt: Securing iot deployments. In *Proceedings of the Applied Networking Research Workshop*, pages 50–57, 2018.

[36] Lora Kolodny. Elon musk emails employees about 'extensive and damaging sabotage' by employee. https://www.cnbc.com/2018/06/18/elon-musk-email-employee-conducted-extensive-and-damaging-sabotage.html, 2018. Accessed: 2019-04-12.

[37] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip A Porras. Delta: A security assessment framework for software-defined networks. In *NDSS*, 2017.

[38] Markus Legner, Tobias Klenze, Marc Wyss, Christoph Sprenger, and Adrian Perrig. Epic: Every packet is checked in the data plane of a path-aware internet. In *29th USENIX Security Symposium*, USENIX Security '20, 2020.

[39] Matt Lepinski and Kotikalapudi Sriram. Bgpsec protocol specification. *Draft-ietf-sidr-bgpsecprotocol*, 2013.

[40] Grace Lewis, Sebastian Echeverría, Craig Mazzotta, Christopher Grabowski, Kyle O'Meara, Amit Vasudevan, Marc Novakouski, Matthew McCormack, and Vyas Sekar. Kalki: a software-defined iot security platform. In *IEEE Virtual World Forum on Internet of Things 2020*, 2020.

[41] Lionel Litty, H Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, 2008.

[42] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *Proc. of USENIX Security*, 2008.

[43] Lee Mathews. Boeing is the latest wannacry ransomware victim. https://www.forbes.com/sites/leemathews/2018/03/30/boeing-is-the-latest-wannacry-ransomware-victim/#9b1382d66344, 2018. Accessed: 2018-11-19.

[44] M. McCormack, S. Chandrasekaran, G. Liu, T. Yu, S. Wolf, and V. Sekar. "security analysis of networked 3d printers". In *IEEE Workshop on the Internet of Safe Things*, 2020.

[45] Matt McCormack, Amit Vasudevan, Guyue Liu, Sebastián Echeverría, Kyle O'Meara, Grace Lewis, and Vyas Sekar. Towards an architecture for trusted edge iot security gateways. In *3rd USENIX Workshop on*

Hot Topics in Edge Computing (HotEdge 20). USENIX Association, June 2020.

[46] Jonathan M. McCune et al. Trustvisor: Efficient TCB reduction and attestation. In *IEEE S&P*, 2010.

[47] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P*, May 2010.

[48] Jad Naous, Michael Walfish, Antonio Nicolosi, David Mazières, Michael Miller, and Arun Seehra. Verifying and enforcing network paths with icing. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, New York, NY, USA, 2011. Association for Computing Machinery.

[49] David Naylor et al. And then there were more: Secure communication for more than two parties. In *CoNEXT*. ACM, 2017.

[50] Phil Oester. Linux kernel memory subsystem copy on write mechanism contains a race condition vulnerability. https://www.kb.cert.org/vuls/id/243144/, 2016. Accessed: 14 February 2020.

[51] Patrick Howell O'Neill. Russian hackers are infiltrating companies via the office printer. https://www.technologyreview.com/f/614062/russian-hackers-fancy-bear-strontium-infiltrate-iot-networks-microsoft-report/, 2019.

[52] Danny Palmer. Iot security: Now dark web hackers are targeting internet-connected gas pumps. https://www.zdnet.com/article/iot-security-now-dark-web-hackers-are-targeting-internet-connected-gas-pumps/, 2019.

[53] Misun Park, Ketan Bhardwaj, and Ada Gavrilovska. Toward lighter containers for the edge. In *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[54] Andriy Pavlovych and Carl Gutwin. Assessing target acquisition and tracking performance for complex moving targets in the presence of latency and jitter. In *Proceedings of Graphics Interface 2012*, GI '12, page 109–116, CAN, 2012. Canadian Information Processing Society.

[55] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, 2015. USENIX Association.

[56] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 201–216, 2018.

[57] Phillip A Porras, Steven Cheung, Martin W Fong, Keith Skinner, and Vinod Yegneswaran. Securing the software defined network control layer. In *NDSS*, 2015.

[58] Dan RK Ports and Tal Garfinkel. Towards application security on untrusted operating systems. In *HotSec*, 2008.

[59] J.M. PORUP. How hacking team got hacked. https://arstechnica.com/information-technology/2016/04/how-hacking-team-got-hacked-phineas-phisher/, 2016. Accessed: 2019-10-02.

[60] Santhosh Prabhu, Gohar Irfan Chaudhry, Brighten Godfrey, and Matthew Caesar. High-coverage testing of softwarized networks. In *Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges*, pages 46–52, 2018.

[61] Daniel Quist, Lorie Liebrock, and Joshua Neil. Improving antivirus accuracy with hypervisor assisted analysis. *J. Comput. Virol.*, 7(2), May 2011.

[62] Daniel Quist, Lorie Liebrock, and Joshua Neil. Improving antivirus accuracy with hypervisor assisted analysis. *Journal in computer virology*, 7(2):121–131, 2011.

[63] S. Raileanu, T. Borangiu, O. Morariu, and I. Iacob. Edge computing in industrial iot framework for cloud-based manufacturing control. In *2018 22nd International Conference on System Theory, Control and Computing (ICSTCC)*, pages 261–266, 2018.

[64] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. ftpm: A software-only implementation of a TPM chip. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 841–856, Austin, TX, August 2016. USENIX Association.

[65] Rattrap. https://www.myrattrap.com, 2018. Accessed: 2018-03-23.

[66] Christian Röpke and Thosten Holz. Preventing malicious sdn applications from hiding adverse network manipulations. In *Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges*, pages 40–45, 2018.

[67] Volker Roth, Wolfgang Polak, Eleanor Rieffel, and Thea Turner. Simple and effective defense against evil twin access points. In *Proceedings of the First ACM Conference on Wireless Network Security*, WiSec '08, page 220–235, New York, NY, USA, 2008. Association for Computing Machinery.

[68] Bob Rudis, Wade Woolwine, and Kwan Lin. 2020: Q2 threat report. https://www.rapid7.com/research/report/2020Q2-threat-report/, 2020.

[69] J. M. Rushby and B. Randell. A distributed secure system. In *1983 IEEE Symposium on Security and Privacy*, pages 127–127, April 1983.

[70] Alex Schiffer. How a fish tank helped hack a casino. https://www.washingtonpost.com/news/innovations/wp/2017/07/21/how-a-fish-tank-helped-hack-a-casino/, 2017. Accessed: 2019-09-23.

[71] Fabian Schwarz and Christian Rossow. SENG, the sgx-enforcing network gateway: Authorizing communication from shielded clients. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[72] Sandra Scott-Hayward. Design and Deployment of Secure, Robust, and Resilient SDN Controllers. In *NetSoft*. IEEE, 2015.

[73] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of SOSP*, 2007.

[74] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao. Are we ready for sdn? implementation challenges for software-defined networks. *IEEE Communications Magazine*, 51(7):36–43, 2013.

[75] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proc. of CCS*, 2009.

[76] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jiseong Noh, and Brent Byunghoon Kang. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 78–89, 2014.

[77] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, et al. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, 2009.

[78] A. K. Simpson et al. Securing vulnerable home iot devices with an in-hub security manager. In *2017 IEEE PerCom Workshops*, 2017.

[79] L. Singaravelu, C. Pu, H. Haertig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *EuroSys*, 2006.

[80] Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Model Checking Invariant Security Properties in OpenFlow. In *ICC*. IEEE, 2013.

[81] https://suricata-ids.org, 2020.

[82] Aisha Syed, Bilal Anwer, Vijay Gopalakrishnan, and Jacobus Van der Merwe. Depo: A platform for safe deployment of policy in a software defined infrastructure. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 98–111, 2019.

[83] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, 2006.

[84] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable.

In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 279–292, 2006.

[85] Hendrik Tews, Tjark Weber, Marcus Völp, Erik Poll, MCJD van Eekelen, and PJB van Rossum. Nova micro–hypervisor verification. *CTIT technical report series*, 2008.

[86] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, pages 1–14, 2018.

[87] uan Pedro Tomás. Smart city case study: Santander, spain. https://enterpriseiotinsights.com/20170116/smart-cities/smart-city-case-study-santander-tag23-tag99, 2017.

[88] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *2013 IEEE S&P*, 2013.

[89] Amit Vasudevan. The uber extensible micro-hypervisor framework (uberxmhf). In *Practical Security Properties on Commodity Computing Platforms*, pages 37–71. Springer, 2019.

[90] Amit Vasudevan and Sagar Chaki. Have your pi and eat it too: Practical security on a low-cost ubiquitous computing platform. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 183–198. IEEE, 2018.

[91] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 87–104, Austin, TX, August 2016. USENIX Association.

[92] Amit Vasudevan, Bryan Parno, Ning Qu, Virgil D. Gligor, and Adrian Perrig. Lockdown: Towards a safe and practical architecture for security applications on commodity platforms. In *Proc. of TRUST*, 2012.

[93] Amit Vasudevan, Bryan Parno, Ning Qu, Virgil D Gligor, and Adrian Perrig. Lockdown: Towards a safe and practical architecture for security applications on commodity platforms. In *International Conference on Trust and Trustworthy Computing*, pages 34–54. Springer, 2012.

[94] Amit Vasudevan, Ning Qu, and Adrian Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. In *Proc. of IEEE HICSS*, 2011.

[95] Haopei Wang, Guangliang Yang, Phakpoom Chinprutthiwong, Lei Xu, Yangyong Zhang, and Guofei Gu. Towards fine-grained network security forensics and diagnosis in the sdn era. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 3–16, New York, NY, USA, 2018. Association for Computing Machinery.

[96] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proc. of EuroSys 2012*, 2012.

[97] Xitao Wen et al. Towards a Secure Controller Platform for OpenFlow Applications. In *HotSDN*. ACM, 2013.

[98] Judson Wilson et al. Trust but verify: Auditing the secure internet of things. In *MobiSys*, pages 464–474. ACM, 2017.

[99] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated bug removal for software-defined networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 719–733, 2017.

[100] F. Xiao, J. Zhang, J. Huang, G. Gu, D. Wu, and P. Liu. Unexpected data dependency creation and chaining: A new attack to sdn. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1512–1526, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[101] X. Xiong, D. Tian, and P. Liu. Practical protection of kernel integrity for commodity os from untrusted extensions. In *Proc. of NDSS*, 2011.

[102] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425, 2015.

[103] C. Yoon, S. Lee, H. Kang, T. Park, S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Flow wars: Systemizing the attack surface and defenses in software-defined networks. *IEEE/ACM Transactions on Networking*, 25(6):3514–3530, 2017.

[104] Miao Yu, Virgil D. Gligor, and Zongwei Zhou. Trusted display on untrusted commodity platforms. In *ACM CCS*, pages 989–1003, 2015.

[105] Tianlong Yu, Seyed Kaveh Fayaz, Michael P Collins, Vyas Sekar, and Srinivasan Seshan. Psi: Precise security instrumentation for enterprise networks. In *NDSS*, 2017.

[106] 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*, pages 1–7, 2015.

[107] Zeek. https://zeek.org, 2020.

[108] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. of SOSP*, 2011.

[109] Peng Zhang, Hao Li, Chengchen Hu, Liujia Hu, Lei Xiong, Ruilong Wang, and Yuemei Zhang. Mind the gap: Monitoring the control-data plane consistency in software defined networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 19–33, 2016.

[110] Peng Zhang, Shimin Xu, Zuoru Yang, Hao Li, Qi Li, Huanzhao Wang, and Chengchen Hu. Foces: Detecting forwarding anomalies in software defined networks. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 830–840. IEEE, 2018.

[111] Wei Zhang, Abhigyan Sharma, Kaustubh Joshi, and Timothy Wood. Hardware-assisted isolation in a multi-tenant function-based dataplane. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[112] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE S&P*, 2012.

[113] Zongwei Zhou, Miao Yu, and Virgil D. Gligor. Dancing with Giants: Wimpy Kernels for On-demand Isolated I/O. In *Proc. of IEEE S&P*, 2014.

A Additional Design Details

A.1 Packet Signing Algorithm Evaluation

As a sign and verify operation is performed twice for every packet, our signing algorithm needs to provide high throughput for short message sizes. We benchmarked a range of potential algorithms, from public-key signatures (e.g., ECDSA) to signed message authentication codes (e.g., HMAC, CMAC, etc.). We determined that a SHA1-HMAC was optimal on the Raspberry Pi platform.

We compared the network throughput when computing a signature in user space for each packet to a baseline throughput when no signatures were calculated (shown in Table 6).

Table 6: Signing algorithm comparison on Raspberry Pi 3B for a user space application signing full MTU packets.

Algorithm	Normalized Throughput	Signature Length (bytes)
Baseline (no signing)	1.0	0
HMAC-SHA1	0.243	20
HMAC-MD5	0.236	16
HMAC-SHA256	0.221	32
CMAC-AES	0.0313	16
CMAC-RC2	0.0274	8
ECDSA	0.00013	72
UMAC*	0.00067	16
GMAC*	0.27	16

*Requires random nonce/IV

A.2 vTPM vs physical TPM

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 7, the virtual TPM was 20x faster while providing 8x more measurement storage locations (PCRs).

Table 7: TPM PCR extend comparison between virtual and physical TPMs on the Raspberry Pi 3B.

TPM	Median Time	PCR Registers
Physical	17.2 milliseconds	24
Virtual	0.86 milliseconds	configurable, up to 120

A.3 Model Evaluation Resources

When evaluating our system model, we measured the system resources required for a given model size. The resource utilization is shown in Table 8.

Table 8: Resources required for Alloy model analysis.

Level of BMC	Variables	Memory (MB)	Time (seconds)
5	18,212	110	0.062
10	81,239	159	0.27
20	430,174	702	1.98
50	4,976,794	4,441	143.63
60	8,276,734	6,034	406.59
100	35,358,494	10,240	5,998.29

A.4 Sample IoT Data

Our sampling of commodity IoT devices consisted of 10 devices. These ranged from smart speakers, to vacuum cleaners and security systems. We measured their network traffic while under normal usage conditions. Their network characteristics are shown in Table 9.

Table 9: etwork characteristic of a sample of home IoT devices during normal usage.

Device Type	Steady-state Throughput (pps)	Average packet size (bytes)	Peak Throughput (pps)	Time between bursts (s)
Speaker A	191	793.5	2502	30
Speaker B	46	287.5	154	*
Vacuum A	8	291.5	>30	>10000
Vacuum B	5	103.5	5	*
Smart Plug A	10	87.5	50	>10000
Smart Plug B	15	166.5	32	>60
Security Sys A	10	87.5	10	*
Security Sys B	28	203.5	32	*
Smart TV	81	185.5	347	>10000
Smart Camera	392	322.5	392	*

* - Device traffic not bursty