2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) | 978-1-6654-4188-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/RTCSA52859.2021.00019

Formalizing an Architectural Model of a Trustworthy Edge IoT Security Gateway‡

Matt McCormack,* Amit Vasudevan,† Guyue Liu,* Vyas Sekar*
*Carnegie Mellon University - CyLab, †Carnegie Mellon Software Engineering Institute

Abstract*—Today's edge networks continue to see an increasing number of deployed IoT devices. These IoT devices aim to increase productivity and efficiency; however, they are plagued by a myriad of vulnerabilities. Industry and academia have proposed protecting these devices by deploying a "bolt-on" security gateway to these edge networks. The gateway applies security protections at the network level. While security gateways are an attractive solution, they raise a fundamental concern: *Can the bolt-on security gateway be trusted?

This paper identifies key challenges in realizing this goal and sketches a roadmap for providing trust in bolt-on edge IoT security gateways. Specifically, we show the promise of using a micro-hypervisor driven approach for delivering practical (deployable today) trust that is catered to both end-users and gateway vendors alike in terms of cost, generality, capabilities, and performance. We describe the challenges in establishing trust on today's edge security gateways, formalize the adversary and trust properties, describe our system architecture, encode and prove our architecture trust properties using the Alloy formal modeling language. We foresee our trustworthy security gateway architecture becoming a practical and extensible formal foundation towards realizing robust trust properties on today's edge security gateway implementations.

I. Introduction

IoT devices are increasingly being deployed into edge environments, from home networks to manufacturing floors. Unfortunately, these devices are plagued by a myriad of vulnerabilities [2], [3], which attackers have leveraged as stepping stones into protected networks and as launch pads for other attacks [4]–[7]. Consequently, these IoT devices pose a continuing threat to the security of our edge networks.

Industry and academia have proposed securing (potentially vulnerable) IoT devices on edge networks with on-site security gateways [2], [8]–[13]. These "bolt-on" security gateways are designed to intercept all traffic to and from an IoT device and apply security protections via *middleboxes* at the *network level* (*e.g.,* a firewall). These middleboxes can be used to implement "network patches" which mitigate a device's vulnerabilities without patching the device's software.

While bolt-on security gateways are gaining popularity and are a deployable solution, they raise a fundamental question: *How do we ensure that the system providing these network level protections is trustworthy?* As an example scenario, consider a smart factory with a plethora of IoT devices protected by multiple security gateways. The factory's security gateways provide network level protections tailored to each individual IoT device's vulnerabilities. Unfortunately, security gateways form a single point of failure. They are particularly vulnerable to an adversary who can compromise the security gateway (by exploiting OS and/or application vulnerabilities), as the gateway typically runs commodity software (*e.g.,* Linux, Docker, OVS, Snort, etc.). Once compromised, an adversary can modify the gateway's protections (*e.g.,* remove a firewall rule) thereby enabling attacks against an IoT device in order to stop/alter production (à la [14], [15]).

Current approaches for securing applications in untrusted cloud environments could potentially be applied to establish trust in security gateways. These approaches rely on hardware-specific capabilities (*e.g.,* SGX [16], [17], MPX [18]). Unfortunately, such approaches have high performance overheads (not practical for IoT deployments) and also lack generality. They are limited to a specific processor class and only support user space applications with constrained memory allocation. Furthermore, addressing security vulnerabilities [19]–[21] requires fabricating newer revisions of the hardware.

At a high level, we envision a *trusted* IoT security gateway architecture, that provides an overarching guarantee that the correct security protections are applied to each IoT device's network traffic at all times, including when under attack (more details in §IV). We use this aforementioned definition of trust throughout this paper. Our architecture aims to provide robust trust properties to a broad range of legacy hardware platforms utilizing existing software with a reasonable performance overhead. There are three challenges to realizing our vision:

- **Formalizing Adversary and Trust Properties (§IV)**: To design a trusted architecture, we need to consider a *rich adversary* model, where the adversary could attack any software component and data in transit. Existing security gateway architectures often utilize a software defined network (SDN) architecture [2], where the data plane enforces network level protections, and the control plane orchestrates these protections to achieve a policy. While prior work on SDN security [22]–[24] explored some attack scenarios, they tackle a limited adversary model, only analyzing a subset of the architecture (*e.g.,* routing, application permissions). Both control and data plane elements and their communications must be protected to achieve trust.
- **Supporting Dynamic Middleboxes (§V)**: The architecture must provide trust in dynamic middleboxes that are constantly being reconfigured (*e.g.,* IoT devices frequently

‡This paper is an extended version of a workshop paper presented in USENIX Hot Topics in Edge Computing (HotEdge) 2020 [1].

978-1-6654-4188-9/21/$31.00 ©2021 IEEE
DOI 10.1109/RTCSA52859.2021.00019

leaving and joining edge networks). Prior work in cloud computing proposed using secure hardware (*e.g.,* SGX, TrustZone), placing entire applications in a trusted execution environment (*e.g.,* enclave). While this approach could prevent tampering, it fails to support today's dynamic middleboxes due to limited available memory (*e.g.,* 128MB on SGX [25]), reduced functionality (*e.g.,* inability to perform system calls [26] required for timestamps), and the high performance costs of changing enclaves (*e.g.,* reducing performance by up to 30% [27], [28]). Furthermore, only placing pieces of the application in an enclave suffers severe performance costs [28]. An ideal solution provides trust to legacy software on any hardware platform in a performant manner.

- **Secure and Efficient Communication (§VI)**: A trusted security gateway requires secure communications, enforcing protections at a per-packet granularity both between and across the control and data planes. Existing tunneling techniques (*e.g.,* IPSec, TLS) could be used between planes, but are too expensive for protecting across a plane (*e.g.,* tunneling a packet between middleboxes on the data plane). Low performance overheads are required for latency-sensitive devices (*e.g.,* real-time, closed-loop robot controllers).

In this paper, we argue that a *micro-hypervisor* based approach is a promising architectural basis for building trust in edge security gateways. A micro-hypervisor, like a traditional hypervisor, is a software reference monitor that provides core security capabilities (*e.g.,* memory isolation, mediation, and attestation) that can be applied to effectively address the aforementioned challenges. In contrast to traditional hypervisors, these capabilities are provided with a dramatically reduced trusted computing base (TCB) and complexity (hence the *micro* prefix) which enable formal verification to rule out potential vulnerabilities [29]–[31]. Furthermore, micro-hypervisors provide an extensible foundation for realizing robust trust properties without a loss of generality and minimal performance overhead [29], [30], [32]–[34]. Last but not least, in contrast to approaches using specific hardware capabilities which limit applications (*e.g.,* SGX's limitations described above), micro-hypervisors can support a variety of hardware platforms (x86 [30], [33], ARM [32], microcontroller [31]) running unmodified software (*e.g.,* Linux) [29], [31], [34]. Thus, a micro-hypervisor provides a practical and secure foundation for building security mechanisms towards realizing our vision.

Our intuition to leverage a micro-hypervisor based approach is motivated by the success micro-hypervisors have had on commodity platforms [29]. However, to the best of our knowledge, micro-hypervisors have not been used in edge IoT gateways. To this end, our contributions are: (1) a more holistic system adversary model and cardinal security properties for an edge IoT security gateway; (2) a high-level architecture based on micro-hypervisors to enable a practical and flexible solution; and (3) a formal model of our IoT security gateway architecture encoded in the Alloy formal modeling language with proofs of foundational security properties. Our model and proofs can be found at: https://github.com/slab14/Gateway_Alloy_Model

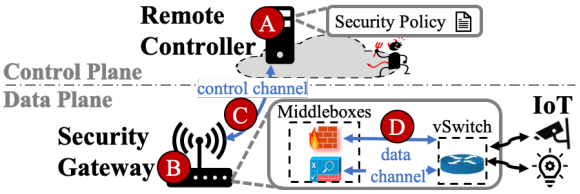

Fig. 1: Attack vectors for bolt-on security architecture.

II. Motivation

Traditional security solutions (*e.g.,* antivirus) fall short for IoT devices due to resource requirements and device heterogeneity [2], [8]. Security gateway based approaches [2], [8], [10]–[13], [35] have been proposed to secure IoT deployments.

"Bolt-on" Security Gateways: At a high level, these approaches insert a security gateway running virtualized middleboxes (*e.g.,* firewall, IDS) to protect deployed IoT devices. To achieve this, the gateway intercepts all traffic to and from the IoT device and sends the network traffic to a middlebox which imposes a security policy (*e.g.,* IoT may not SSH).

While initially these security gateways employed a single monolithic middlebox running a static configuration (*e.g.,* an IDS with a default ruleset), recent work [2], [8], [35] highlighted the need for isolated (*e.g.,* each device has its own set of middleboxes), device-specific (*e.g.,* each middlebox configured to protect a specific device's vulnerabilities) middleboxes that support dynamic security policies (*e.g.,* changing based upon context, such as other device's status). The need for these new capabilities has increased the complexity of the security gateway architectures (shown in Fig. 1), adding virtual switches (vSwitch) for routing data to the appropriate middleboxes and a remote controller for dynamically configuring each gateway's protections (*e.g.,* middlebox configurations, vSwitch routes) to achieve the security policy.

These "bolt-on" gateways are promising for securing IoT deployments; however, they are currently untrusted. Under attack, these security gateways could become ineffective, or even worse, become a launchpad for new attacks.

Motivating Scenario: An attacker could launch attacks at multiple points in the architecture (shown in Fig. 1). For example, an attacker could: (1) use an unpatched exploit [36], [37] to compromise the gateway itself (B in Fig. 1) and (2) modify the middlebox configuration such that it allows the attacker's traffic to pass through to enable the attacker to compromise a factory's IoT device and steal proprietary data (à la [7]). Beyond modifying the software, an attacker could also tamper with network messages. For example, modifying packets on the data channel between the vSwitch and the middlebox (D in Fig. 1), redirecting traffic to the wrong middlebox, evading security inspections.

A trusted security architecture needs to protect the gateway and controller's software while prohibiting tampering with network traffic. We look to prior work for potential solutions.

TABLE I: Limits of prior piecemeal solutions providing end-to-end trust in software-defined security architectures.

Approach	Mitigates	Limitations
Trusted hardware [16], [17]	Attacks A & B	Reduced performance & deployability
Path authentication [38]–[40]	Attacks C & D	Must trust software, protect secret keys

Limitations of Prior Work: Prior work has looked at securing individual pieces of this bolt-on software-defined security architecture, but lacks end-to-end solutions (reference Table I). In order to protect middleboxes from being modified, recent work in securing middleboxes in untrusted cloud environments (*e.g.,* [16], [17]) placed middlebox in a trusted enclave (*e.g.,* SGX, TrustZone). This could protect the middleboxes on the gateway from modification (*e.g.,* blocking Attack B in 1). Unfortunately, these do not provide hollistic protection and create high performance costs for communicating between enclaves (*e.g.,* placing the middlebox and vSwitch in separate enclaves). Research on securing the controller (*e.g.,* [41], [42]) has been limited to protecting the control plane from malicious applications and ensuring consistency between the control and data planes. Unfortunately, none provide runtime protections against an attacker capable of compromising the OS.

Strawman Solution: A natural strawman solution would be to run the gateway and controller software in an enclave, with a secure tunnel (*e.g.,* IPSec, TLS) protecting the control channel. This solution has been used for securing middleboxes from untrusted cloud providers [16], [17].

While this solution could prevent tampering with the gateway and controller software and protects control messages, it requires specific hardware and has three key limitations. First, only limited applications are supported. Applications running inside an enclave have limited memory access (*i.e.,* 128MB for SGX) and can only perform user space actions (*e.g.,* no system calls). Second, there is a significant performance overhead for initiating communication with an enclave, which is magnified if multiple enclaves must be utilized (*e.g.,* isolating multiple middleboxes in a chain), impacting low-latency edge devices. Third, vulnerabilities identified in trusted hardware may require long timelines to patch. This approach would be sufficient for a single, static protection on the gateway; however, the need to support dynamic middleboxes which are isolated and constantly changing entails a different approach.

Ideally, we want a solution that can be deployed on a wide range of hardware platforms, including resource-constrained edge platforms. Further, it needs to support existing software applications, while adding minimal performance overhead.

III. Trusted Security Gateway Architecture

We envision a *trusted, extensible, and widely-deployable edge security gateway architecture* that addresses the security challenges of today's edge IoT deployments. When fully realized, our architecture would enable new trustworthy "security-as-a-service" offerings that providers (e.g., edge ISPs, CDNs, IoT providers) could offer to IoT consumers, ensuring the correct security protections are applied at all times. For instance, this architecture could provide a trusted mechanism for enforcing IoT security best practices (*e.g.,* access-control policies in a device's Manufacturer Usage Description specification [43]).

System Assumptions: To scope our design space, we make three assumptions about our trusted architecture.

- *Only IP traffic:* 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.
- *Gateway is the first-hop:* all packets to and from an IoT device must go through the gateway as their first-hop. An attacker cannot directly access an IoT device or use an evil twin attack (*e.g.,* [44]) to bypass the gateway.
- *Correct middlebox implementation:* device-specific protections are realized by a single middlebox. This middlebox is capable of running multiple network functions (*e.g.,* a firewall and a proxy in the same middlebox). We assume that these middleboxes are implemented correctly, able to detect and block all network exploits targeting the IoT device they are protecting. Further, they drop all packets not to the IoT device they are protecting.

We can consider a strawman design space categorized along two axes. First, approaches dependent on hardware functionality (*e.g.,* [16], [17]) are limited in both the hardware platforms and software they can support. Additionally, their security properties rest on a complex and opaque implementation in microcode and silicon [45], known to have vulnerabilities [19]–[21]. Second, pure software approaches (*e.g.,* formal verification, secure programming languages) are limited as they require significant reimplementation and verification effort. As many commonly used software applications on edge security gateways span over 100,000 lines of C/Java this quickly becomes intractable.

We argue that it is dangerous to tie critical security features to either hardware implementations that require new hardware to address threats, or to software approaches that require significant reimplementation or formal verification effort of the entire software stack. Instead we advocate leveraging legacy hardware features in combination with a small TCB and extensible software framework to provide our fundamental trust properties and protect edge devices from evolving threats.

Consequently, we make a case for a *micro-hypervisor* based approach to enable a trusted edge security gateway architecture (Fig. 2) that allows *retrofitting* security protections *to only the necessary* system components. A micro-hypervisor is in essence a software reference monitor [46], that acts as a guardian, implementing access control to system resources (*e.g.,* files, sockets) using a small TCB. These protections can be applied in a fine-grained manner, protecting a single data value (*e.g.,* secret key) or a complex set of objects (*e.g.,* virtual machine) with minimal performance overhead. Micro-hypervisors provide a strong foundation for fine-grained mediation, isolation, and attestation with a small TCB [29], [30], [33], [47], which allows for security services to be designed and implemented as extensions [29]. Due to their simplicity and small TCB, micro-hypervisors are amenable

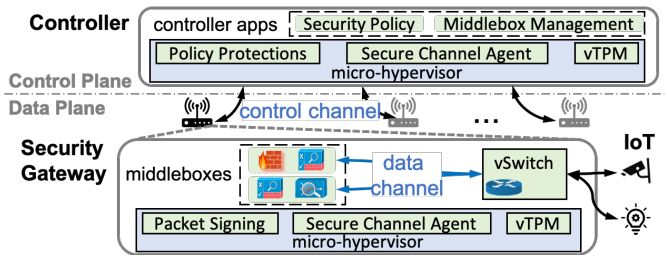

Fig. 2: High-level view of a trusted, extensible, and widely-deployable edge security gateway architecture.

to formal verification for ruling out potential vulnerabilities within their code [29], [31]. Additionally, micro-hypervisors can potentially be supported on any hardware platform (*e.g.,* x86 [30], [33], ARM [32], custom microcontroller [31]).

We build on top of the aforementioned micro-hypervisor enabled foundational capabilities to construct our trusted security gateway architecture. The controller and gateway's software run on top of a micro-hypervisor, allowing us to support any commodity OS and application stack. On the controller, we migrate critical data (*e.g.,* the security policy) into micro-hypervisor extensions to isolate it from untrusted software (*e.g.,* the OS). Further, all access is mediated by the micro-hypervisor, prohibiting an attacker from subverting the data's integrity. On the gateway, we assign a set of customized middleboxes to each device and isolate these from each other. Additionally, we periodically measure the signature of each middlebox and the vSwitch to verify their integrity. Finally, the controller and the gateway run *trusted agents*, which are micro-hypervisor extensions used to mediate communication between the control and data planes, to ensure the instantiated protections correctly reflect the security policy.

An edge security gateway architecture built atop a micro-hypervisor provides three key benefits:

- *Fine-grained security:* It provides fine-grained isolation and mediation which allow for precisely ensuring that the architecture enforces the correct protections while being performant.
- *Extensible:* It is extensible allowing for rapid growth of new security functionality and response to emerging threats.
- *Deployable:* It is widely-deployable, supporting a wide range of hardware platforms (*e.g.,* x86, ARM) while utilizing existing software, with a low performance overhead for providing trust.

Challenges: For our architecture to provide a holistic defense against end-to-end attacks, it must address three challenges:

- **Necessary Security Properties (§IV):** Identify the security properties that ensure trust under a holistic adversary model. These properties guide the design of our architecture.
- **Support Dynamic Middleboxes (§V):** Enable protecting the dynamic middleboxes required by an IoT environment, ensuring an adversary cannot modify the protections.
- **Secure Communications (§VI):** Provide per-packet protections with a low performance impact, guaranteeing an adversary cannot modify or spoof packets.

We begin by formally defining our trust properties.

TABLE II: Holistic adversary capabilities, generated using the STRIDE model (referencing Fig. 1's attack vectors).

Example Threat	Vector	Violates
Modify controller software (*e.g.,* security policy)	A	P_{sw1}, P_{sw2}
Modify gateway software (*e.g.,* middlebox config)	B	
Spoof control channel message (*e.g.,* vSwitch route)	C	P_{com1}
Tamper with data channel message (*e.g.,* skip middlebox)	D	P_{com2}

IV. Adversary and Trust Properties

Our goal is to provide end-to-end protections against a holistic threat model. Within the SDN domain, this would entail protecting both the control and data planes (*e.g.,* from BGP hijacking [48], [49]). However, prior works on IoT security gateways have typically only considered a narrow threat model.

To this end, we systematically define such an adversary, with a goal of inhibiting the gateway's protections (*i.e.,* enable exploiting a protected IoT device). We assume our adversary has knowledge of the security architecture as well as network access to all devices. We group our adversary capabilities into two categories: (1) ability to compromise a device's software stack (*i.e.,* software on the controller or gateway; A, B in Fig. 1), and (2) ability to inject/modify network messages (*i.e.,* the control, data channels; C, D in Fig. 1).

We use the STRIDE threat modeling tool [50] to generate a set of adversary capabilities (summarized in Table II), that inhibit the architecture's ability to protecting an IoT device. While not a complete list, we use it to define the fundamental security properties needed for our trusted architecture.

Based upon our adversary model (Table II), we posit that there are a minimum of five fundamental properties required for a trusted security gateway architecture.

- **Software Integrity** (P_{sw1}): Ability to detect code and data modifications (*e.g.,* changes in middlebox configuration).
- **Data Isolation** (P_{sw2}): Ability to isolate security critical logic (*e.g.,* keep the OS from accessing the security policy).
- **Data Mediation** (P_{sw3}): Ability to have a trusted entity mediate access to security critical data (*e.g.,* blocking an untrusted application's access to secret keys).
- **Secure Control Channel** (P_{com1}): Ability to trust data transferred between the controller and gateway (*e.g.,* the gateway only executes commands from the controller).
- **Secure Data Channel** (P_{com2}): Ability to that trust packets are routed through the correct middleboxes (*e.g.,* packets should not be processed by a wrong middlebox).

To guide and inform the design of our trustworthy architecture, we create a formal model of today's software-defined IoT security gateway architectures. This model helps us identify critical components and interfaces of our architecture, formally define and encode the corresponding architectural elements, and prove desired trust properties.

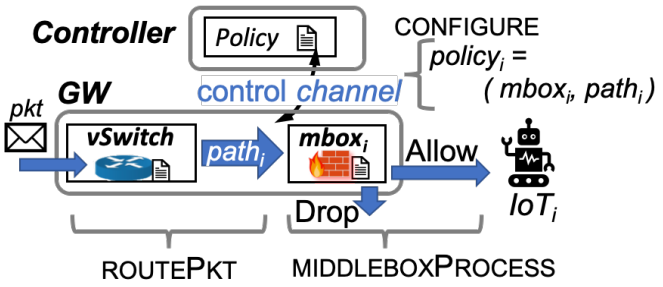

Fig. 3: Key components of JETFIRE architecture model.

A. Background on the Alloy Modeling

We build a formal model of bolt-on IoT security architectures using the Alloy modeling language [51] (see Listing 1). 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 [51].

The Alloy model is compiled into a scope-bounded satisfiability problem and analyzed by off-the-shelf SAT solvers. This analysis can be used to identify if an instance of the model exists and to identify counterexamples to constraints. We use this analysis to identify attack vectors, refine our security architecture, and to formalize our trust properties (§IV-C).

B. Model Description and Semantics

Our bolt-on software-defined IoT security architecture model (seen in Fig. 3) 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 single gateway to explain our abridged Alloy model in Listing 1 (the full model can be found in [52]).

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 signatures (*i.e.,* each Controller has one Policy, line 2). The controller maintains the security policy and uses the control channel to configure each gateway based on the policy. Each gateway is managed by the associated controller. The gateway receives its policy over the control channel and installs paths in the vSwitch and then instantiates the middleboxes. Each path specifies which middlebox a specific IoT device's traffic should be routed through.

Then we model how the gateway processes packets using Alloy's `fun` 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 (ROUTEPKT, line 12). Then the middlebox processes the packet and determines if the packet is benign or malicious (MIDDLEBOXPROCESS, lines 13). Packets that are benign are routed back to the switch and then sent out to the IoT device while all other packets are dropped.

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

```
1: sig Controller {
2:     policy : one Policy,
3:     cntrlChannel: one Channel
4: }

5: sig Gateway {
6:     vswitch : one vSwitch,
7:     mbox : set Middlebox,
8:     controller : one Controller,
9:     cntrlChannel: 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:     c == g.controller
21:     TamperProof(c.policy)
22:     SecureChannel(g.cntrlChannel, c.cntrlChannel)
23:     RemoteAttest(g.controller)
24:     RemoteAttest(g.vswitch)
25:     for g.mbox_i in c.policy do
26:         RemoteAttest(g.mbox_i)
27:         AuthenticateRoute(g.vswitch, g.mbox_i)

28: assert ProcessPktCorrectly(g : Gateway, pkt : Packet)
29:     TrustedGateway(g)
30:     pkt ∈ BenignPkts ⟹ ProcessPkt(pkt, g) == Allow
31:     pkt ∈ MaliciousPkts ⟹ ProcessPkt(pkt, g) == Drop
```

Next, we define a trusted gateway architecture using Alloy's `pred` interface (lines 19-27). 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 an architecture to be trusted. First, an attacker must not be able to tamper with the policy on the controller (TAMPERPROOF is true in line 21). Second, the control channel between the controller and the gateway must be secure so that it is immune to an attacker injecting malicious messages (SECURECHANNEL is true in line 22). Third, the correct software must be running on the controller, vSwitch, and middlebox (CORRECTSW is true for lines 23-26). Finally, each packet must follow the path specified by the controller and enforced by the vSwitch and each middlebox (AUTHENTICATEROUTE is true in line 27). If all of these conditions are true then the gateway architecture is trusted.

Finally, we define our goal that all output packets were processed correctly using Alloy's `assert` interface (lines 28-31). In Alloy, an `assert` claims that a series of statements must be true based upon the model, and will generate a counterexample if any of the claims do not hold to be true. A trusted gateway architecture can achieve this goal. Specifically, it allows all benign packets while dropping all malicious packets.

Listing 2 Alloy model of policy isolation and mediation.

```
1: pred TAMPERPROOF(obj : Object)
2:     ISOLATEDMEMORY(obj)
3:     MEDIATEDACCESS(obj)

4: assert TRUSTWORTHYPOLICY(c : Controller)
5:     TAMPERPROOF(c.policy)
```

Listing 3 Alloy model of instance validation.

```
1: pred REMOTEATTEST(obj : Object)
2:     CORRECTSOFTWARE(obj)
3:     ATTESTCORRECTNESS(obj)

4: assert CORRECTINSTANCE(c : Controller)
5:     REMOTEATTEST(c)

6: assert CORRECTINSTANCE(g : Gateway)
7:     REMOTEATTEST(g.vswitch)
8:     for g.mbox_i in g do
9:         REMOTEATTEST(g.mbox_i)
```

C. Security Properties

Using our architectural model and semantics, we formally define our overarching security property and its sub-properties.

Overarching Security 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, so that benign packets are allowed and malicious packets are dropped (modeled in Listing 1). This can be denoted as:

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

To achieve this property in the presence of an attacker, we must ensure the gateway architecture is trusted, expressed formally:

$$\begin{aligned} TrustedGateway(GW, Controller) \iff \\ tamperProof(policy) \wedge \\ correctInstance(GW, Controller) \wedge \\ secureChannel(GW, Controller) \wedge \\ \forall mbox_i,\ authenticateRoute(vSwitch, mbox_i) \end{aligned} \tag{2}$$

$$\begin{aligned} \text{Where: } GW = \{channel, vSwitch, \{mbox_0, \ldots, mbox_n\}\} \\ Controller = \{channel, policy\} \end{aligned}$$

We realized Equations 1 and 2 in Alloy, as shown in Listing 1. Where we instantiate a gateway architecture with a controller and gateway that processes packets using a software middlebox. The PROCESSPKT function determines if each packet is either benign and allowed to be forwarded to its final destination or malicious and dropped. The TRUSTEDGATEWAY predicate tests if an architecture (composed of a gateway and a controller) is trustworthy. Where it verifies that the controller is assigned to the gateway, that the controller's security policy is tamper proof, the channel between the controller and gateway is secure, that the controller, vswitch, and all middleboxes is attested to be correct, and that each packet being processed follows an authenticated route. A gateway meeting these requirements ensures that all benign packets are allowed and malicious packets are dropped (confirmed with the assert statement).

We further decompose our overarching security property into four sub-properties. These sub-properties can be broadly grouped into two categories: (1) protecting data and running code (P_{sw1}, P_{sw2}) and (2) protecting network communications (P_{com1}, P_{com2}). Next, let's look at each of them in detail.

Security Policy Isolation and Mediation (P_{sw1}): The first sub-property is to protect the security policy stored in the controller. Controller applications are subject to attacks [53], [54] 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 and all access requires mediation by a trusted entity (modeled in Listing 2). Such defenses block the OS and other untrusted applications from accessing and modifying the security policy. This can be denoted as:

$$\begin{aligned} tamperProof(policy) \iff \\ isolatedMemory(policy) \wedge mediatedAccess(policy) \end{aligned} \tag{3}$$

Listing 2 depicts how we modeled Equation 3 in Alloy. The TAMPERPROOF predicate checks if an object is located in isolated memory and all accesses are mediated. The assert checks that a controller's policy is tamper proof.

Component Instance Validation (P_{sw2}): Besides the security policy, the software of key components must not be altered by an attacker (*e.g.,* Attack B in 1 where the middlebox was altered [16], [17]). To achieve this, we need to validate that the correct instance of key components is running. This includes validating the controller, vSwitch and all middleboxes (modeled in Listing 3). Where each middlebox instance is the type and configuration (*e.g.,* rule set) currently specified by the controller's security policy. This can be denoted as:

$$\begin{aligned} correctInstance(GW, Controller) \iff \\ remoteAttest(Controller) \wedge \\ remoteAttest(vswitch) \wedge \\ \forall mbox_i,\ remoteAttest(mbox_i) \end{aligned} \tag{4}$$

We model our implementation of Equation 4 in Listing 3. The predicate REMOTEATTEST checks that an object has the correct software and its correctness can be attested. We then verify this by asserting it for the controller, virtual switch, and each middlebox.

Control Message Integrity and Authentication (P_{com1}): To protect against control channel attacks (*e.g.,* Attack C in 1) [22], [23], [53]–[55], 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

Listing 4 Alloy model of message integrity and authentication.

```
1: pred TRUSTWORTHYCHANNEL(ch : Channel, k : Key)
2:     AUTHENTICATEDENCRYPTED(ch)
3:     TAMPERPROOF(k)

4: assert SECURECHANNEL(c : Controller, g : Gateway)
5:     SAMECHANNEL(c.cntrlChannel, g.cntrlChannel)
6:     TRUSTWORTHYCHANNEL(c.cntrlChannel, c.cntrlChannel.key)
7:     TRUSTWORTHYCHANNEL(g.cntrlChannel, g.cntrlChannel.key)
```

Listing 5 Alloy model of packet path and data validation.

```
1: function GETPATH(pkt : Packet, p : Policy)
2:     mbox = GETMBOXFROMPOLICY(p, pkt)
3:     path = {vswitch, mbox, vswitch}
4:     return path

5: pred AUTHENTICATEHOP(pkt : Packet, h : HopLocation)
6:     UNMODIFIEDDATA(pkt)
7:     CORRECTHOPLOCATION(h)

8: assert CORRECTPATH(pkt : Packet, path : Path, p : Policy)
9:     path.vswitch, path.mbox in GETPATH(p, policy)

10: assert AUTHENTICATEPKTROUTE(pkt : Packet, p : policy)
11:     path = GETPATH(pkt, p)
12:     CORRECTPATH(pkt, path, p)
13:     for hop in path do
14:         AUTHENTICATEHOP(p, hop)
```

controller can send middlebox configuration commands to the gateway). Meanwhile, the secret keys used by the channel are isolated and any access is mediated by a trusted entity (modeled in Listing 4). This can be denoted as:

$$\begin{aligned} &secureChannel(channel) \iff \\ &authenticatedEncrypted(channel) \wedge \\ &isolatedMemory(keys) \wedge mediatedAccess(keys) \end{aligned} \tag{5}$$

A model of our Alloy implementation of Equation 5 is shown in Listing 4. The TRUSTWORTHYCHANNEL predicate checks that the channel (*i.e.,* method being use to send messages between the controller and the gateway) uses authenticated encryption and that the encryption keys are tamper proof. The assert then verifies that the channel between the gateway and the controller uses authenticated encryption and both hosts protect the encryption keys.

Packet Path and Data Validation (P_{com2}): 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 [38], [39]. We aim to provide similar guarantees in order to detect packets maliciously routed to the wrong middlebox (*e.g.,* Attack D in 1). In particular, we need to verify whether the intended path of a packet has been enforced, and whether packet data has been modified (modeled in Listing 5). This can be denoted:

$$\begin{aligned} &authenticateRoute(vswitch, mbox_i) \iff \\ &\forall pkt,\ intendedPath(pkt, policy) = mbox_i \implies \\ &actualPath(pkt) = vswitch \rightarrow mbox_i \rightarrow vswitch \end{aligned} \tag{6}$$

TABLE III: Security sub-properties mitigate example attacks.

Attack in Fig. 1	Security Properties	Solutions
Attack A	P_{sw1}	Limit access to security critical operations
Attack B	P_{sw2}	Verify component matches expected
Attack C	P_{com1}	Blocks control channel injections
Attack D	P_{com2}	Identifies packet path modification

Listing 5 depicts our Alloy implementation of Equation 6. The function GETPATH provides the mapping of packets to the appropriate middlebox, and returns the packet's sequence of hops on the gateway. Next, we define the predicate AUTHENTICATEDHOP which checks to ensure the packet was not modified and that it arrived a the correct hop location. We verify that packets being processed by the gateway follow authenticated routes using the assert statements to ensure the packet follows the correct path and that the packet is not modified at each hop location.

A system that provides these properties will *by construction* stop the example attacks in §II. As shown in Table III, security policy isolation and mediation (P_{sw1}) blocks an attacker from modifying the security policy (Attack A in 1). Component instance validation (P_{sw2}) enables the architecture to detect an attacker modifying a middlebox (Attack B in 1). Control message integrity and authentication (P_{com1}) blocks an attacker from being able to inject malicious control messages. Finally, packet path and data validation (P_{com2}) mitigates local attackers modifying packets (Attack D in 1). Further details on our formal model can be found in [52].

We envision our architecture supporting additional properties, but focus on these as fundamental to a trusted architecture. These fundamental properties can be grouped into: (1) protecting running code (P_{sw}) and (2) protecting communications (P_{com}). Next, we discuss our approach for providing these.

V. SUPPORTING DYNAMIC MIDDLEBOXES

Ideally, the entire codebase on both the control and data planes could be robustly protected from an attacker. However, we view this as impractical as it either incurs significant performance costs (*e.g.,* multiple enclaves to process each packet) or requires significant reimplementation (*e.g.,* migrating 100,000+ lines of C/Java). Instead we look to apply fine-grained security properties to the portions of the codebase that impact the architecture's protections, thereby creating a robustness against our adversary (§IV). Specifically, we apply periodic, remote attestation to guarantee the code's integrity (providing P_{sw1}). This allows the code to run with minimal performance degradation, while bounding the duration it is vulnerable to attack. Additionally, critical code (*e.g.,* security policy) can be protected with a hypervisor extension to isolate it (providing P_{sw2}) and mediate access to it (providing P_{sw3}).

Periodic, Remote Attestation: IoT security gateways rely upon a large codebase to provide device-specific protections. We look to prior work in remote attestation, such as Trusted Platform Modules (TPM) [56], [57], in order to precisely

guarantee that the appropriate software stack is running (providing P_{sw1}). Upon boot, the correct baseline software stack, composed of the micro-hypervisor, OS, and critical software components (*e.g.,* controller, vSwitch, middleboxes, etc.) is verified. Subsequently, new modules that will impact the provided protections (*e.g.,* a new middlebox's code prior to loading) are attested, thereby allowing the architecture to ensure that the correct protections are instantiated.

During runtime, we periodically re-attest critical modules (*e.g.,* controller, vSwitch, middleboxes) ensuring an attacker has not tampered with them. For example, the middlebox code must be run outside of the hypervisor to enable high packet throughput. To bound the potential impact of an attacker tampering with this code (*i.e.,* the protection not being applied), the controller remotely attests critical software components on the gateway at the end of every epoch, where the epoch duration can be adjusted to provide a trade-off between the vulnerable window's length and the security overhead.

We leverage a virtual trusted platform module (vTPM) on the micro-hypervisor to provide this attestation capability. A vTPM is a software implementation of a physical TPM and provides many of the same capabilities [57]. Specifically, we leverage its ability to securely store a chain of measurements, by extending a program control register (PCR), and securely providing those stored values (*i.e.,* a PCR quote). These two capabilities enable determining if a software stack on a local or a remote machine matches a known configuration. These vTPM measurements can be applied at a fine granularity, with separate storage for multiple measurements.

Protecting the Controller's Security Policy: While attestation can provide significant guarantees about a code's integrity, there are some pieces of code that merit further protection (*e.g.,* code impacting decisions about the protections provided by the security gateway). Our micro-hypervisor approach enables selectively isolating this code (P_{sw2}) and requiring that access to it be mediated by a trusted entity (P_{sw3}). Examples of such pieces of code are the secret keys used to establish a secure control channel between the planes and the security policy on the controller.

As a concrete example, consider the controller's security policy. The security policy is critical to ensuring the correct protections are implemented (*e.g.,* modifying it could result in the gateway's middleboxes not protecting the IoT devices). This code can be extracted from the controller and placed into memory isolated by the micro-hypervisor (providing P_{sw2}). Further, access to this memory is mediated by the micro-hypervisor's code white-listing (providing P_{sw3}), to ensure that only the controller's code can access the security policy. This combination prohibits an attacker in control of the OS from accessing and modifying hypervisor protected pieces of code, without requiring significant changes to existing code.

VI. Secure and Efficient Communication

Our security architecture requires trust guarantees on both the control channel (between controller and gateway, P_{com1}) and the data channel (along the gateway's packet processing path, P_{com2}). We leverage the micro-hypervisor to provide isolation and mediation to secure these communication channels.

Secure Control Channel: It is crucial that communication between the control and data plane can be trusted as these messages often impact the security protections provided by the gateway. We look to bolster the guarantees provided by traditional tunneling (*e.g.,* IPsec/TLS) between the controller and the gateway to ensure a compromised controller or gateway cannot send spoofed messages over the tunnel (*e.g.,* malicious middlebox configuration commands). To protect these communications (P_{com1}), we leverage a trusted agent pair running in the micro-hypervisor to mediate these communications (*e.g.,* access the secret keys required to send data over this channel). There is an agent on the controller and a corresponding agent on the gateway, which together are responsible for mediating access to the secure channel.

Secure Data Channel: On the data plane, the security protections are dependent upon each packet being processed by the correct middlebox. We can build on prior work on routing path verification (*e.g.,* control plane [38], data plane [39]), to provide per-hop guarantees with a low performance overhead. Specifically, our goal is to guarantee that packets follow the correct path and are processed by the correct sequence of middleboxes on the data plane (P_{com2}). While traditional tunnels could be established between each middlebox and the vSwitch, this would result in significant overhead and processing delays. To protect the data channel, we propose leveraging the micro-hypervisor to enforce the correct path (*i.e.,* middlebox chain) for each packet. We achieve this by having the micro-hypervisor sign and verify each packet along its processing path, dropping packets that fail verification. Our approach differs from prior per-hop authentication proposals (*e.g.,* [38], [39]) as packets remain on a single host where a hypervisor can maintain secret keys.

These digital signatures create a connection between the raw packet data and the middlebox processing the packet, by the secret key (protected by the micro-hypervisor) shared between the vSwitch and each middlebox. Furthermore, the digital signatures can be trusted, as the secret keys are kept in isolated memory only accessible by the micro-hypervisor's mediation, stopping an attacker from forging signed packets.

VII. Model Evaluation

We updated our Alloy model (§IV) to reflect our trusted architecture and analyze its performance. We probed our architecture's model for instances that allow an attacker to violate our trust property and output a malicious packet (*i.e.,* one containing an exploit).

The Alloy Analyzer was unable to identify a counter example resulting in our architecture outputing a packet processed by an incorrect middlebox. It employs bounded model checking (BMC), where models are evaluated up to a bound, N, of each `sig` in the model. We evaluated our model with up to a bound of 100 (*i.e.,* 100 instances of each `sig`) and noted the resources required to show that this analysis

TABLE IV: 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

could be performed on a personal computer (see Table IV). Additionally, we systematically removed constraints related to our security sub-properties (*e.g.,* middlebox software does not need to be correct) and verified that each resulted in a counter example where our overarching security property was violated. This analysis gave us confidence in our our security properties and architecture's design.

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 allow a packet to bypass the middlebox without being detected. Similarly our Alloy model highlighted software components that either needed to be trusted or be regularly attested in order to trust the architecture's operation. Beyond just the middlebox processing the packet, both the virtual switch and the controller software need to be trusted to ensure correct operation.

We noted that our JETFIRE architecture could have applicability beyond the edge IoT security gateway use case and be used to prevent broader SDN attacks.

VIII. CONCLUSIONS

In this paper, we described our overarching vision for enabling a trusted IoT security gateway architecture that is practical and deployable on today's edge networks. We argued that a micro-hypervisor based approach provides robust trust properties while remaining performant and preserving platform generality. Our preliminary implementation on a Raspberry Pi 3 has provided encouraging results with acceptable operational latency. We are currently working on a full end-to-end implementation and evaluation of our security architecture.

ARCHITECTURE MODEL AVAILABILITY

The Alloy model sources for our trustworthy IoT gateway security architecture can be found at:

https://github.com/slab14/Gateway_Alloy_Model

ACKNOWLEDGMENTS

This work was supported in part by NSF award CNS-1564009. This work was also supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This research was also supported by the Carnegie Mellon University (CMU) Manufacturing Futures Initiative, made possible by the Richard King Mellon Foundation. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center (DM21-0728).

REFERENCES

[1] M. McCormack, A. Vasudevan, G. Liu, S. Echeverría, K. O'Meara, G. A. Lewis, and V. Sekar, "Towards an architecture for trusted edge iot security gateways," in *3rd USENIX Workshop on Hot Topics in Edge Computing, HotEdge 2020, June 25-26, 2020*, I. Ahmad and M. Zhao, Eds. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/hotedge20/presentation/mccormack

[2] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. 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*, 2015, pp. 1–7.

[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] J. Porup, "How hacking team got hacked," https://arstechnica.com/information-technology/2016/04/how-hacking-team-got-hacked-phineas-phisher/, 2016.

[5] P. 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.

[6] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *USENIX Security 17*. Vancouver, BC: USENIX Association, 2017.

[7] A. 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.

[8] R. Ko and J. Mickens, "Deadbolt: Securing iot deployments," in *Proceedings of the Applied Networking Research Workshop*, 2018.

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

[10] D. Barrera, I. Molloy, and H. Huang, "Standardizing iot network security policy enforcement," in *DISS 2018*, 2018.

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

[12] "Rattrap," https://www.myrattrap.com, 2018, accessed: 2018-03-23.

[13] "Cujo," https://www.getcujo.com, 2018, accessed: 2018-03-23.

[14] S. Belikovetsky, M. Yampolskiy, J. Toh, J. Gatlin, and Y. Elovici, "dr0wned – cyber-physical attack with additive manufacturing," in *11th USENIX Workshop on Offensive Technology (WOOT 17)*. Vancouver, BC: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/belikovetsky

[15] L. 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.

[16] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, "Safebricks: Shielding network functions in the cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[17] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "Shieldbox: Secure middleboxes using shielded execution," in *Proceedings of the Symposium on SDN Research*, 2018, pp. 1–14.

[18] W. Zhang, A. Sharma, K. Joshi, and T. Wood, "Hardware-assisted isolation in a multi-tenant function-based dataplane," in *Proceedings of the Symposium on SDN Research*, 2018, pp. 1–7.

[19] MITRE, https://nvd.nist.gov/vuln/detail/CVE-2017-5691, 2017.

[20] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[21] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[22] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. A. Porras, "Delta: A security assessment framework for software-defined networks." in *NDSS*, 2017.

[23] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks." in *NDSS*, 2015.

[24] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated bug removal for software-defined networks," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[25] T. D. Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont, "Everything you should know about intel sgx performance on virtualized systems." *POMACS*, 2019.

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

[27] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for intel sgx enclaves," in *Proceedings of the 19th International Middleware Conference*, 2018, pp. 201–213.

[28] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves." in *NDSS*, 2017.

[29] A. Vasudevan, "The uber extensible micro-hypervisor framework (uberxmhf)," in *Practical Security Properties on Commodity Computing Platforms*. Springer, 2019, pp. 37–71.

[30] 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.

[31] M. Ammar, B. Crispo, B. Jacobs, D. Hughes, and W. Daniels, "Sμv - the security microvisor: A formally-verified software-based security architecture for the internet of things," *IEEE Trans. Dependable Sec. Comput.*, vol. 16, no. 5, 2019. [Online]. Available: https://doi.org/10.1109/TDSC.2019.2928541

[32] A. Vasudevan and S. 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)*. IEEE, 2018.

[33] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. 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*, 2009.

[34] H. Tews, T. Weber, M. Völp, E. Poll, M. van Eekelen, and P. van Rossum, "Nova micro–hypervisor verification," *CTIT technical report series*, 2008.

[35] T. Yu, S. K. Fayaz, M. P. Collins, V. Sekar, and S. Seshan, "Psi: Precise security instrumentation for enterprise networks." in *NDSS*, 2017.

[36] P. 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.

[37] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. 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*, 2015, pp. 414–425.

[38] M. Lepinski and K. Sriram, "Bgpsec protocol specification," *Draft-ietf-sidr-bgpsecprotocol*, 2013.

[39] J. Naous, M. Walfish, A. Nicolosi, D. Mazières, M. Miller, and A. Seehra, "Verifying and enforcing network paths with icing," in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2079296.2079326

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

[41] P. A. Porras, S. Cheung, M. W. Fong, K. Skinner, and V. Yegneswaran, "Securing the software defined network control layer." in *NDSS*, 2015.

[42] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, "Rosemary: A robust, secure, and high-performance network operating system," in *Proceedings of the 2014 ACM CCS*, 2014.

[43] E. Lear, R. Droms, and D. Romascanu, "Manufacturer usage description specification," *IETF draft*, 2017.

[44] V. Roth, W. Polak, E. Rieffel, and T. Turner, "Simple and effective defense against evil twin access points," in *Proceedings of the First ACM Conference on Wireless Network Security*, ser. WiSec '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 220–235. [Online]. Available: https://doi.org/10.1145/1352533.1352569

[45] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[46] J. M. Rushby and B. Randell, "A distributed secure system," in *1983 IEEE Symposium on Security and Privacy*, April 1983, pp. 127–127.

[47] J. M. McCune *et al.*, "Trustvisor: Efficient TCB reduction and attestation," in *IEEE S&P*, 2010.

[48] X. Hu and Z. M. Mao, "Accurate real-time identification of ip prefix hijacking," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, 2007, pp. 3–17.

[49] X. Shi, Y. Xiang, Z. Wang, X. Yin, and J. Wu, "Detecting prefix hijackings in the internet with argus," in *Proceedings of the 2012 Internet Measurement Conference*, ser. IMC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 15–28. [Online]. Available: https://doi.org/10.1145/2398776.2398779

[50] A. Shostack, "Experiences threat modeling at microsoft." *Workshop on modeling security (ModSec)*, 2008.

[51] D. Jackson, "Alloy: A new technology for software modelling," in *Tools and Algorithms for the Construction and Analysis of Systems*, J.-P. Katoen and P. Stevens, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 20–20.

[52] "Jetfire code," https://github.com/slab14/Gateway_Alloy_Model, 2020.

[53] 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*, vol. 25, no. 6, pp. 3514–3530, 2017.

[54] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. 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*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 3–16. [Online]. Available: https://doi.org/10.1145/3243734.3243749

[55] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 1512–1526. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00017

[56] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, 2011.

[57] W. Arthur, D. Challener, and K. Goldman, *A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security*. Berkeley, CA: Apress, 2015, ch. History of the TPM, pp. 1–5. [Online]. Available: https://doi.org/10.1007/978-1-4302-6584-9_1