SODA: A Software-defined Security Framework for IoT Environments

SODA: A Software-defined Security Framework for IoT Environments

Yeonkeun Kim[a,*], Jaehyun Nam[a,*], Taejune Park[a], Sandra Scott-Hayward[b] and Seungwon Shin[a,**]

[a]KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea
[b]Queen's University Belfast, University Road, Belfast, BT7 1NN, Northern Ireland, United Kingdom

ARTICLE INFO

Keywords:
IoT Security;
Software-defined Networking;
Network Function Virtualization;
Access Control

ABSTRACT

The Internet of Things (IoT), based on interconnected devices, enables a variety of elegant new services that could not be realized in a traditional environment, and many of these services harvest the information of a potentially sensitive and private nature belonging to individual users. Unfortunately, existing security functions used to protect such information are difficult to implement in an IoT environment due to the widely varying capacities, functionalities, and security requirements of IoT devices. In this work, to protect against unrestricted accesses to other devices and information extortion from these devices, we propose SODA, a secure IoT gateway that enables a device-side dynamic access control and is capable of deploying various security services to protect sensitive and private information. To show its effectiveness and practicality, we assume that a large number of IoT devices are crowded around an IoT gateway, and we implement a prototype of SODA for such an environment based on software-defined-networking (SDN) and integrate virtual network functions (VNFs) over network function virtualization (NFV) on top of a real IoT device. From our evaluation, we demonstrate how SODA mitigates real-world attacks through its security functions, and presents how it satisfies the performance requirements of a real environment.

1. Introduction

The Internet of Things (IoT) is an ecosystem in which devices (i.e., Things) are connected to a network (i.e., the Internet) such that these devices can communicate and share data with each other through the connected network. As this ecosystem provides powerful network connectivity of diverse IoT devices, it is rapidly increasing in popularity; this trend can be easily observed across industry and academia [7, 16, 18, 23, 34, 45]. For instance, smart watches are available today and already selling well [5]. Some IoT services have already been provided by commercial vendors (e.g., Apple Homekit [4] or Samsung SmartThings [35]).

However, an IoT environment that is composed of highly connected IoT devices also introduces critical *security* problems. For example, a compromised or malicious device can easily access other connected devices through an IoT network due to the lack of authentication and access control, and then be used to conduct various attacks on nearby devices, such as stealing private information from the devices. Thus, for those IoT devices that handle sensitive data (e.g., health information), the security problems of the IoT environment could be even more critical.

Security challenges in an IoT environment are often regarded as more complicated than those of legacy networks due to the diversity of IoT devices. In terms of security policies, IoT users have different security requirements for IoT devices; thus, each IoT device should maintain multiple security policies to satisfy their requirements. Unfortunately, those policies can cause unexpected policy and intent conflicts with each other. In addition, those policies can cause some other conflicts when multiple IoT devices access and control the same resources. With regard to security functionalities, when deploying security functions for IoT devices, we need to allocate resources to operate those functions efficiently. However, an IoT device is, in general, a lightweight, single-board computer with low computation power, which means that its functionality is restricted. Hence, we need to carefully consider how to operate necessary security functions with restricted resources as well. Although some pioneering researchers have proposed solutions for securing IoT environments [6, 17, 31, 34, 47, 48], none of them have addressed the combination of these security challenges.

Today's network architecture for an IoT environment is broadly classified into two types: (i) peer-to-peer and (ii) centralized. In the former case, each IoT device directly connects to another device to share information; the AllJoyN project [23] is a good example for this case. In the case of the centralized type, IoT devices are connected to a centralized point (i.e., a gateway) for network services. While the peer-to-peer type is often used in these days, most vendors have adopted the centralized type because it can easily manage large-scale IoT devices through a centralized gateway that handles all network flows among the devices [7, 18]. Following today's trend, we focus on a *centralized* IoT network to address security challenges in an IoT environment.

In this paper, we present a novel security framework, called SODA, to realize centralized security policy and service management for IoT environments (specifically, for the IoT environment where a large number of IoT devices are crowded around a centralized IoT gateway). In our proposal, all IoT devices are connected to SODA, and all security policies (e.g., who can communicate with whom) are controlled by SODA. In addition, SODA provides the capability of

*The first two authors contributed equally to this work.
**Corresponding author

yeonk@kaist.ac.kr (Y. Kim); namjh@kaist.ac.kr (J. Nam); taejune.park@kaist.ac.kr (T. Park); s.scott-hayward@qub.ac.uk (S. Scott-Hayward); claude@kaist.ac.kr (S. Shin)
ORCID(s): 0000-0002-3558-9041 (Y. Kim); 0000-0001-8907-5495 (J. Nam); 0000-0003-1421-5996 (T. Park); 0000-0002-0330-1963 (S. Scott-Hayward); 0000-0002-1077-5606 (S. Shin)

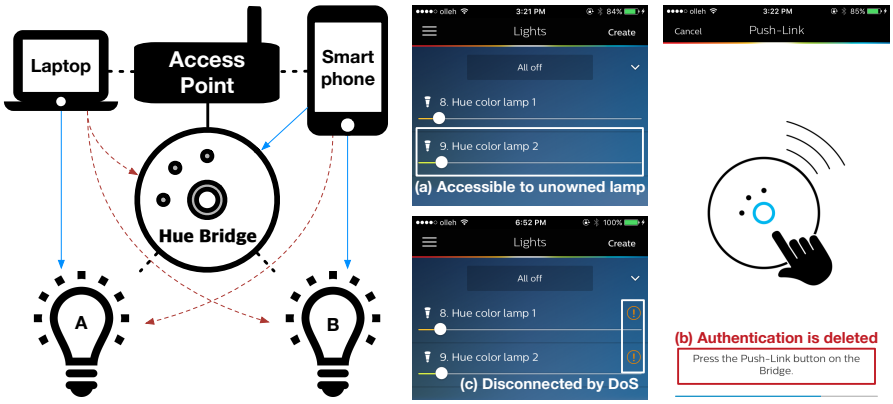


Figure 1: An IoT network with commercial IoT devices - Blue and red dotted lines refer to valid and invalid accesses respectively.

deploying various security services, such as intrusion prevention system and botnet mitigation, and each user (or IoT device) can dynamically take some of security services on request. Security services supported by SODA can be classified into two types: (i) network access control and (ii) network security functions. First, SODA controls (either blocks or allows) network accesses among IoT users and devices according to dynamically defined security policies. Second, SODA provides more complex security capabilities. For example, let us assume that a malicious device (perhaps infected by an attacker) tries to compromise other linked devices. To prevent this, a network intrusion detection system should be activated and be available to target devices in the IoT network. SODA identifies this kind of a requirement and provides relevant security services using virtual network functions realized in SODA.

To implement a prototype of SODA, we leverage software defined networking (SDN) to control network flows among IoT devices [24], and network function virtualization (NFV) [10] to realize security functions inside of a centralized gateway. Then, we demonstrate how SODA can effectively and efficiently protect an IoT network through the prototype of SODA, and our evaluation results show that SODA can successfully enforce user-defined security policies while minimizing conflicts among policies and operating diverse network security functions with reasonable performance.

This paper is organized as follows. Section 2 discusses IoT security issues with an example of commercial IoT devices, and Section 3 presents a new security framework that addresses those issues while its core functionality is elaborated in Section 4. Section 5 describes its prototypical implementation, and Section 6 and 7 demonstrate its effectiveness and practicality. Section 8 reviews the previous studies related to IoT security. Section 9 and 10 conclude the paper while providing some leads for future works.

2. Motivation

This section introduces four security issues in an IoT environment with real-world attack scenarios against *Philips Hue smart lamp* [9, 26]. For these, as shown in Figure 1 (left), we have constructed a simple IoT network to control Hue smart lamps. In this network, two IoT devices (a laptop and a smartphone) are connected to a Hue bridge through an access point (AP), and two Hue smart lamps (denoted as L_A and L_B) are linked to the bridge. In terms of authority, the lamps L_A and L_B are controlled by the laptop and the smartphone respectively, while the bridge is only managed by the smartphone. Now, we elaborate each security issue and lessons learned from the issue.

Limited Access Control. In these days, most IoT devices such as sensors and actuators (e.g., smart bulbs) have limited access controls against users and devices in the same network. For example, Philips Hue bridge is basically linked to an internal network while a smart lamp is connected to the Hub bridge through the ZigBee protocol. Hence, the accesses from the users and devices in external networks would be naturally restricted. However, inside of an internal network, the bridge is controlled by a Hue application through open APIs with unencrypted JSON messages. Thus, anyone in an internal network can easily see the communications between authorized users and IoT devices, and freely create malicious messages to control internal IoT devices (e.g., turning a smart bulb on/off and changing its brightness).

In our IoT network, the smartphone is suppose to only control the lamp L_B. However, in reality, it can also access and control the lamp L_A, as depicted in Figure 1 (a), by simply launching a Hue application. Furthermore, this kind of a situation can be more serious when the laptop that has no authority to control the Hue bridge tries to control the bridge not the lamp L_A. For example, if the laptop constructs and sends a command to get a list of *usernames*, which identifies the authenticated devices that are connected to the bridge, it would receive all usernames (including the smartphone's username) stored in the bridge without any restriction. Then, based on the leaked information, the laptop can delete the smartphone's username via *an open Hue API*, and the smartphone would be de-authenticated, as shown in Figure 1 (b).

Missing Security Intent. Although the issue of access control could be complemented by enforcing user intents, a further issue would arise. Here, let us assume that security policies for the smartphone are enforced into the AP;

blocking any access from the laptop to the bridge while allowing any access from the smartphone. In this case, since the network has no way to figure out the semantics of the bridge access (e.g., accessing the bridge or accessing the lamps through the bridge), it cannot either permit a benign access from the laptop to the lamp L_A or prevent an invalid access to L_A from the smartphone. From this issue, we find that it is important to recognize the semantics of network accesses among IoT devices, and more fine-grained network control should be applied.

Policy Conflict. In addition, there could be some conflicts between policies enforced by IoT elements due to different interests or purposes. Returning to the example network in Figure 1, what if the laptop enforces a policy to access the Hue bridge to control L_A while the smartphone enforces a policy to block the access from the laptop to the bridge? Likewise, two different policies may be enforced for the same resource, and one policy may be overwritten by the other, resulting in incorrect and unintended policy enforcements. For this reason, we need to consider device-side access control policies defined by user intents for security and analyze conflicts among these policies.

Lack of Security Service. As IoT devices are connected to a network, they are easily exposed to network threats without proper security services. For example, when the laptop sends burst commands to turn the lamps on and off within a short time, the connections between the bridge and the lamps can be disrupted (broken), as shown in Figure 1 (c), and IoT users can no longer control the lamps. To prevent such situation, security services (e.g., DoS detector or intrusion detection system) should be integrated with the Hue bridge. However, the reality is that it is difficult to deploy proper security services into each IoT device due to the lack of resources, requiring a means of offloading security services into additional devices that have powerful resources.

Our approaches. The above scenarios demonstrate most critical security issues in IoT networks, and our main goal is to eliminate such issues. Clearly speaking, we (i) collect and enforce user intents in an IoT environment for preserving privacy and securing IoT devices, (ii) recognize the semantics of network accesses and apply more fine-grained network control, (iii) detect and resolve conflicts between different security policies, and (iv) protect an IoT environment against network attacks using security functions within a reasonable resource limit. *Note that our approaches focus on security policy enforcement and security function management rather than authorization or encryption solutions, which have been discussed in previous studies [1, 2, 29].*

3. SODA Design

In this section, we introduce a new security framework, called SODA, for an IoT environment, and describe how SODA manages IoT users and devices while realizing the advanced security for these elements with a low cost. The design of SODA focuses on achieving a user-defined security scheme, which is specialized for an IoT network where each element has different security policies, and a software-based security scheme, which is implemented on a flexible and programmable device. Figure 2 illustrates the overall architecture of SODA. Here, we elaborate how each component in SODA realizes the security policy and service management in an IoT environment.

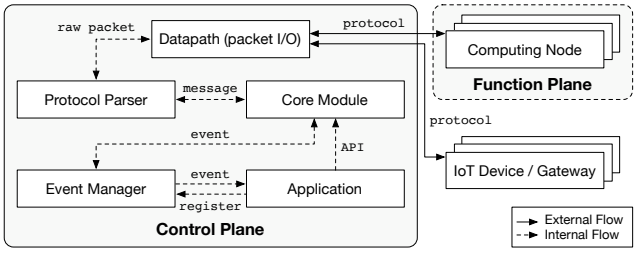


Figure 2: The overall architecture of SODA

Table 1
The list of SODA Events (*Abstracted Event)

Event	Description
`ElemRegister`*	New network element is successfully registered in SODA.
`ElemDeregister`*	Registered element is deregistered.
`ElemUpdate`*	Registered element is updated.
`PolicyEnforce`	New policy is submitted to SODA.
`PolicyConflict`	Submitted policy conflicts with installed policies.
`PolicyInstall`	Submitted policy is successfully installed in the network.
`PolicyRevoke`	Installed policy is revoked.

3.1. Control Plane

The control plane of SODA manages network elements and states (e.g., sessions and user-defined security policies) through five internal components. *Datapath* (i.e., packet I/O) first provides a communication interface between the control plane and the function plane and between the control plane and the data plane where IoT devices are connected. Then, *Protocol Parser* deals with control messages delivered from the function and data planes and forwards them to the core module for further process. *Core module* plays a major role in the control plane, and it especially (i) manages network elements and states, (ii) provides base security functions, and (iii) detects and resolves security policy conflicts. We will separately elaborate the details of this core module in the next section (Section 4). *Event Manager* enables event-driven communication between internal components (including applications). *Applications* are built with a programmable interface, and process delivered events for various IoT services.

Event-driven Model. SODA employs an event-driven model to invoke specific components in the control plane. For this, each component registers its own event handler to receive and process desired events. SODA provides two types of events (element-related and policy-related events), and Table 1 presents the events that it currently provides.

Table 2
The list of SODA Northbound APIs (*Abstracted API)

API	Description
`EraseElement(ElemID)`*	Erase stored element information from a local table.
`LoadElementContent(ElemID)`*	Load element information stored in a local table.
`UpdateElementContent(ElemID, ElemContent)`*	Update element information stored in a local table.
`EnforcePolicy(OwnerID, PolicyContent)`	Install a submitted security policy and return an ID, if successful.
`RevokePolicy(PolicyID)`	Revoke an installed security policy in a policy table.
`SendMsgToElement(CompID, Message)`	Send an application message to an element.

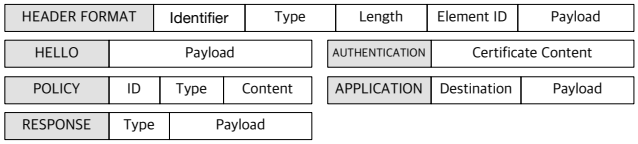


Figure 3: The header format and types of the SODA Protocol

In terms of element-related events, all elements (e.g., user devices, network devices, and applications) should be registered to join into an IoT network managed by SODA. For example, if a new user device requests to join into the network, SODA first verifies whether the user device is authorized or not (see the Session Manager in Section 4.2). Then, the core module raises a `DeviceRegister` event if the authorization process is successful. Similarly, when applications or network devices join to SODA, *register events* would be issued. To synchronize some information of registered elements with SODA, an element reports the updated information to SODA through *update events*. For instance, the core module raises a `SwitchUpdate` event when a switch reports some updated information such as flow statistics.

For user-defined security policies, SODA provides four special events under its base security functionality. Unlike network states, security policies can cause a critical problem if they are not processed correctly. Thus, SODA generates several *policy events* for any changes in policy management, allowing applications to know if user policies have been either accepted or rejected. We will discuss the detail of the policy management in Section 4.1.

Programmable Interface. Rather than direct accesses among internal components or stored states, SODA provides a *programmable interface* between applications and the core module to prevent abnormal resource accesses and to enhance flexible usability.

Table 2 describes the northbound APIs of SODA. First, top three northbound APIs enable applications to read and write the element information in the local tables of the core module. Similar to element events, in this paper, we present abstracted APIs rather than showing all APIs for each element to reduce complexity, meaning that each element (a user device and a computing node) has three dedicated northbound APIs (e.g., `LoadDeviceContent()` API to read some information about a specific device, `LoadApplicationContent()` to read some information about a specific application, and a `LoadSwitchContent()` to read some information about a specific switch). To avoid unexpected policy conflicts [27, 28], SODA also provides two policy APIs (`EnforcePolicy()` and `RevokePolicy()`), and these APIs ensure that security policies are correctly processed before policy installation. Besides them, the last API allows applications to send messages to specific elements for application-level functionalities.

3.2. Function Plane

In general, IoT devices have limited resources because they are built on small, low-power and low-performance hardware. Thus, it is hard to deploy full security features applied in commodity hardware on each IoT device. For this reason, previous studies have introduced a scheme that offloads heavy security functions to powerful computing devices [46]. Motivated by these studies, SODA constructs a pool of computing nodes, called a *function plane*; multiple computing nodes perform security functions offloaded from IoT devices by employing NFV techniques, which dynamically support virtualized network functions for various security services.

If some network elements are able to directly enforce two kinds of policies for satisfying their functional requirements and for mitigating existing threats into the function plane, these policies can contend with each other since SODA cannot distinguish between them. Thus, SODA maintains the information of detected threats from offloaded security services in a local table. Then, it makes sure that the policies to mitigate security threats always have higher priorities than those for the requirements of the elements.

As computing nodes are independent services from the core functionalities of SODA, the function plane can be either collocated with the control plane together or separated from the control plane. For such flexibility, SODA utilizes its own protocol instead of internal events to relay the communication between computing nodes and the control plane. Each computing node has a client to interact with the control plane, and follows the same steps to join into a SODA network as a user device. By default, the function plane is collocated with the control plane together.

3.3. SODA Protocol

In order for the control plane to communicate with remote elements (i.e., the function plane and the data plane that contains IoT users and devices), SODA provides its own protocol, named SODA protocol. Figure 3 illustrates the header format and message types of the protocol. The first field of the protocol header contains an identifier to indicate

that a message follows the specific version of the SODA protocol. The second field represents a message type, which is one of five message types (i.e., HELLO, AUTHENTICATION, POLICY, APPLICATION, and RESPONSE), as shown in Figure 3. The third field has the length of a message. The fourth field contains a random number that is assigned by SODA to each element, allowing SODA to perform the source validation and identification of an incoming message.

Handshake Process. An element (e.g., device) first needs to send a HELLO message to initiate a connection with the control plane of SODA, and then SODA checks if the message contains a correct version of the SODA protocol. If checked, SODA sends back the response for the HELLO message. SODA does not allow any non-authenticated element to join into an IoT network; thus, the element should send an AUTHENTICATION message that contains the details about itself to SODA (see the Session Manager in Section 4.2). When the authentication process is done, SODA generates a random unique number and assigns it to the element as a unique element ID, and confirms that the element has successfully received the assigned ID (by acknowledgment). Finally, it establishes the connection of the element and raises an ElemRegister event to inform the subscribers of the new element. After that, if any subsequent messages from the element do not contain the assigned element ID, SODA regards them as malicious, dropping them immediately.

Policy Installation. After establishing the connection, the element can apply its policies to register user intents about the desired security functionalities from the network. The policies are submitted through POLICY messages, and SODA raises PolicyEnforce events to inform its submissions to components in the control plane. If SODA detects any conflicts between submitted policies and installed policies, it triggers PolicyConflict events (see the PolicyManager in Section 4.1). After running a resolution mechanism with PolicyConflict events, the results of the resolution are forwarded to a policy enforcer (i.e., components or applications) via RESPONSE messages. Consequently, all policies that pass the resolution process are installed into the network with PolicyInstall events. SODA raises a PolicyRevoke event when a policy is removed. Then, the deleted policy with the reason of its deletion is notified to a policy enforcer through a RESPONSE message.

Information Delivery. SODA supports an application-level communication, which means that an application can provide more flexible functionality within the network. This communication is based on APPLICATION messages, which contain an identifier of a target element and a content. Thus, when SODA receives an APPLICATION message, it parses the message and forwards its content to the target element. If there is no matched element, it would notify a failure to an event trigger (a sender) using a RESPONSE message.

4. Core Module

In this section, we particularly elaborate the core module of SODA that plays a pivotal role in SODA. The core module of SODA is, as depicted in Figure 4, internally composed of three submodules: *policy manager*, *session manager*, and *NFV manager*.

4.1. Policy Manager

When an element applies its policy as an intent, the policy is first delivered to the *policy manager* in the core module. Then, the policy is processed through four functions (i.e., role-based policy prioritization, policy conflict detection, policy conflict resolution, and policy compiler) in the policy manager. In this work, we follow a generic access control policy scheme, which consists of a source, a destination, an access type and an access control decision (action):

$$\text{Source} \rightarrow \text{Destination} : \text{Data}, \text{Type} \mid \text{Action}$$
$$\text{where } type = \{read, write\} \text{ and } action = \{allow, deny\}$$

Role-based Policy Prioritization. The policy manager rescales the priority of a submitted policy according to the role of a policy enforcer (i.e., an application) since a policy enforcer may intentionally set a high priority for its policy. For this, SODA classifies given policies into three types of roles (i.e., admin, security, and user roles) based on who enforces the policies [27, 28].

More specifically, policies enforced by a network administrator are classified as an *admin role*, which has the highest priority to guarantee that those policies are not overwritten by any other policies. The second level is a *security role*, which has a higher priority than other policies except for those tagged with the admin role. The security role is assigned to security functions that are running on the network to protect the network from various network-oriented threats. The last one is a *user role*, which has the lowest priority in the network. As the name of the role implies, it is assigned to policies applied from connected elements, and most policies are classified as this role.

Policy Conflict Detection. This mechanism is designed to avoid unexpected security problems caused by policy conflicts, as motivated by previous studies [27, 28]. Once a submitted policy is rescaled by the role-based policy prioritization, this policy conflict detection checks if there is any conflict between the submitted policy and already-installed policies. For this, it searches a policy table to find installed policies that contain any overlapped access domain with the submitted policy.

Furthermore, SODA provides a *policy domain extension*, which prevents detouring the detection mechanism by abusing installed policies. As elements in an IoT environment automatically exchange their resources to perform specific functions, some resources can be copied to different elements, which can cause a security problem since a network may not have a security policy for the copied resources, allowing undesired accesses to them. To prevent this problem, the detection mechanism extends a domain of a policy that targets a resource copied to a different element. As a result, the copied resource can be correctly managed as well.

Policy Conflict Resolution. Rather than simply notifying administrators (or users) to fix policy conflicts, SODA

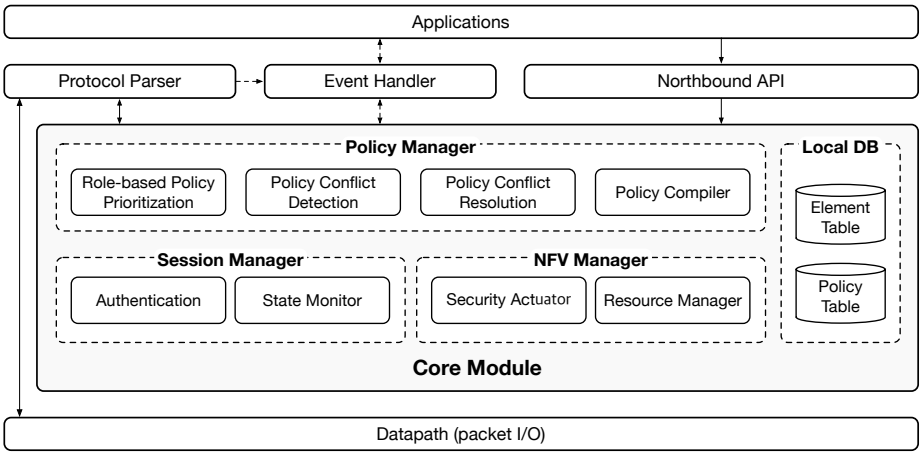


Figure 4: The internal composition of the core module

automatically resolves detected conflicts. For this, SODA first separates the domain of conflicted policies to minimize the disturbance of user intents, and these *policy domain fragments* enable the non-overlapped domain of conflicted policies to be at least enforced successfully. In the case of overlapped domains, SODA follows the priority of a policy, meaning that policies with higher priorities would be applied while rejecting conflicted policies with lower priorities. For instance, as the security role has a higher priority than the user role, policies that mitigates network threats would be applied in preference to user policies. After resolving conflicts, finalized policies are stored in the policy table of the core module, and these policies with their detailed information are notified to the policy owner through a `RESPONSE` message.

Policy Compiler. When a new access between IoT elements occurs in the network, the data plane requests an access policy (whether this access is allowed or not) to the control plane. Then, once the request comes to the control plane, the policy compiler first converts the low-level flow information to the form of a high-level policy, and looks up its policy table to find a policy matched to this. If SODA has a matched policy, the policy compiler generates a low-level flow rule corresponding to the policy, and sends the flow rule back to the data plane. If there is no matched policy, SODA regards the access as abnormal and blocks the access.

4.2. Session Manager

To join into the IoT network managed by SODA, an element has to interact with a *session manager*, which maintains the states of elements (e.g., the connectivities of the elements).

Authentication. During the handshake process, the certificate of a new element is passed to the session manager and verified to determine whether the element should be allowed or blocked. Currently, a pre-shared key between an element and SODA is used to decrypt the certificate, but it can be replaced to another authentication method such as a trusted third-party or key sharing. Then, once the connection is established, the information of the connected element is stored in an internal table, and used to validate the sources of network flows in the network and to convert security policies to low-level flow rules applied in the data plane.

State Monitor. The session manager also monitors the states of elements to detect when an element leaves a network. For each connected element, it periodically sends a `HELLO` message as a heartbeat, and raises an `ElemDeregister` event if there is no response from an element within a timeout. This timeout value is defined by network administrators and requires a trade-off between network overhead and fast detection.

4.3. NFV Manager

In Section 3.2, it is mentioned that the function plane communicates with the core module in the control plane to mitigate detected threats in the network. When a security function running on a computing node detects a network attack, the detail of the attack is passed to an *NFV manager* through an `APPLICATION` message. Furthermore, each computing node reports the current statistics of system resources to this module to avoid resource-related failures.

Security Actuator. When an offloaded security function detects a network attack, it sends the details of the attack to the security actuator of the NFV manager. As the detected attack is expressed as a low-level network flow, the security actuator first generates a mitigation policy from the attack flow using the policy compiler. If there is no conflict between the mitigation policy and highly prioritized policies, the security actuator stores it in a policy table and installs the flow rule at the data plane to block the detected attack.

Resource Manager. Although SODA may have relatively powerful resources than most IoT devices, it can still be faced with resource-related failures unless resource management is in place [36]. For this reason, each computing node in SODA reports its current system resource usage to this resource manager at fixed intervals. The resource manager calculates the resource distribution across computing nodes based on collected resource statistics, maximizing resource utilization. Currently, SODA uses max-min fairness

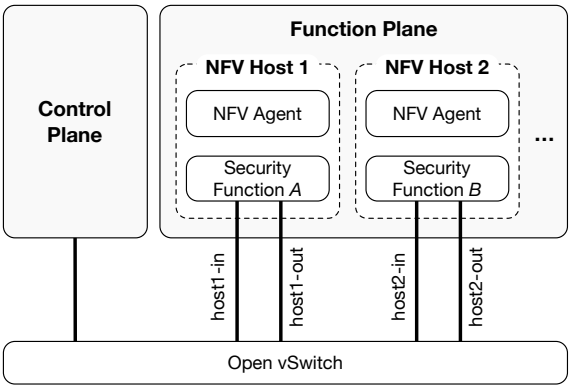


Figure 5: The implementation diagram of SODA's prototype

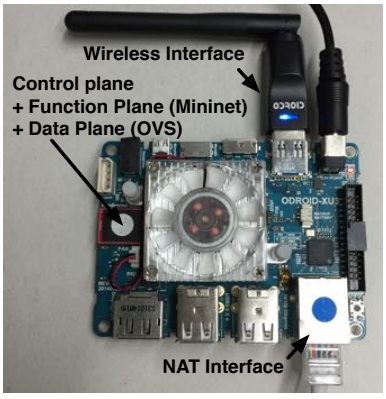


Figure 6: The prototype of SODA on a single-board device

for distributing system resources to computing nodes.

5. Implementation

Here, we describe how the design of SODA is developed on top of a real device. As shown in Figure 5, a prototype of SODA is mainly composed of three components: i) a control plane, ii) a function plane, and iii) a data plane.

For the control plane of SODA, we have first devised a new lightweight SDN controller from scratch in C (referred from [20, 21]), and the data plane of SODA has been implemented by modifying Open vSwitch (OVS) 2.9.1 [15], which is the most popular software switch in these days. In terms of the SODA protocol, we have built it as an application-layer protocol in the modified OVS to simplify the implementation of the data plane by using a flow table provided by OVS. Then, the control plane communicates with the data plane to handle network flows among IoT devices using control messages based on the SODAprotocol. For high-performance message handling, asynchronous I/O (`epoll`), a thread pool with multiple workers, and `glib` hash tables [44] have been adopted inside of the control plane. To provide diverse security functions under limited resources, we have constructed a function plane by leveraging Mininet 2.2.1 [14], which supports a lightweight process-based virtualization, rather than simply adopting full virtualization (e.g., virtual machines)

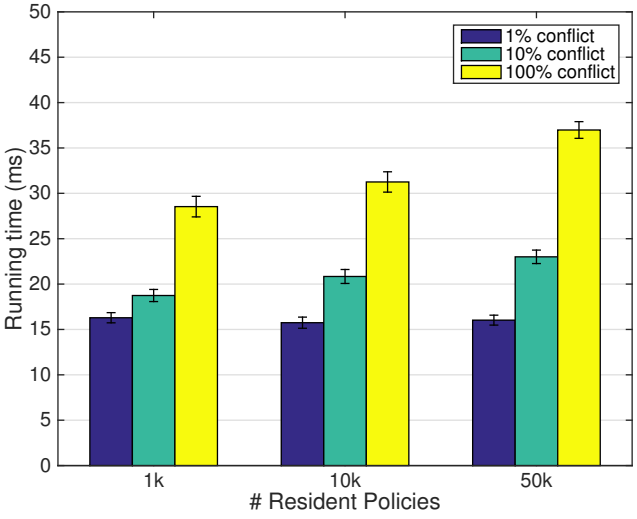


Figure 7: Policy processing times to detect and resolve policy conflicts with the different number of policies in SODA

that requires a large amount of resources. For flexible resource management, we have implemented a NFV client in Python and deployed it into each NFV host (i.e., a computing node) to communicate with the control plane and control the resources allocated to security functions.

The prototype of SODA with all of the components has been installed on top of Ubuntu 16.04 (Linux kernel v4.9) and deployed in an Odroid-XU3 board [13], which is one of single board devices (e.g, Raspberry Pi [30]) that have similar capabilities to legacy IoT gateways, as shown in Figure 6. Also, we have installed and modified a Realtek wireless device driver to have the board service as a controllable WiFi network.

6. Microbenchmark

In this section, we evaluate the performance overhead caused by our system by measuring the computation times of our policy management scheme with different number of policies, and the performance (i.e., throughput and latency) of our system with different number of IoT devices.

6.1. Testbed Environment

We have built a testbed comprising of two Odroid-XU3 devices [13] to evaluate the processing overhead of SODA itself. For this, one of the devices is deployed as an IoT gateway where our prototype implementation is installed. The other device is used as a IoT client, which is directly linked to the data plane of SODA to generate various requests with the identities of multiple IoT devices and feeds the generated traffic into SODA. Also, we have defined security policies that allow the generated traffic to pass through the testbed (network) using a simple policy-based forwarding application running on SODA.

6.2. Policy Management Overhead

To see how much the performance overhead our policy management scheme causes, we evaluate the execution time

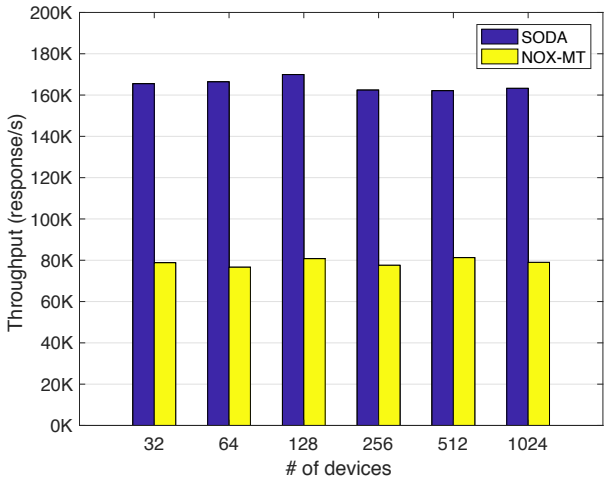


Figure 8: Throughput comparison of SODA with NOX-MT

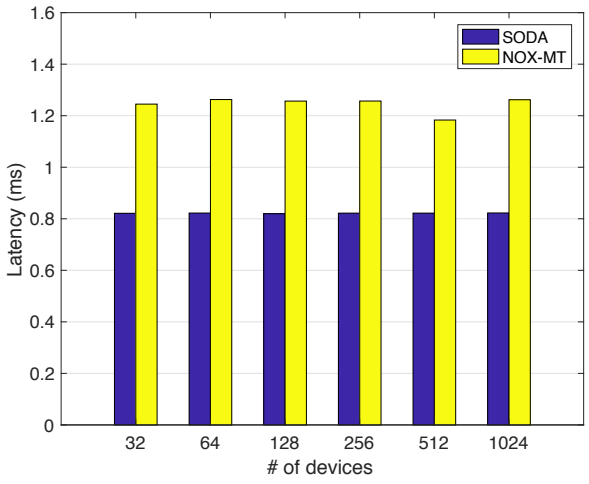


Figure 9: Latency comparison of SODA with NOX-MT

of the policy conflict detection and resolution processes. For this, we generate a large number of random policies in the policy-based forwarding application, and measure the time to process the generated policies. Since the processing time of our mechanisms depend on the volume of policies submitted into SODA, we vary the number of policies and the rate of policy conflicts among them.

Figure 7 shows the execution time to detect and resolve policy conflicts when hundred new policies are submitted. We vary the number of resident (already-installed) policies at the policy table in the core module from 1K to 30K and the rate of conflicts from 1% to 100% (i.e., all newly enforced policies conflict with a portion of resident policies). In all cases, SODA takes less than 40ms to detect and resolve policy conflicts. Here, we observe that the number of installed policies has a minor impact on the overall execution time, but the execution time increases as the rate of policy conflicts increases. This is because of the reconstruction of the policy table. During the resolution process, some of entries that contains conflicted policies in the policy table are reconstructed according to the resolution results. Thus, the more submitted policies are conflicted with resident policies, the more the reconstruction time increases. However, we argue that such processing time does not affect the data transmission among IoT devices since it only occurs once as soon as a new device connects to the network.

6.3. Performance Evaluation

Throughput Measurement: To measure the throughputs between IoT devices, we have modified the Cbench tool [37], which is widely used to evaluate the performance of SDN controllers, and the modified version of the tool randomly generates requests with the identities of various IoT devices instead of original PACKET_IN messages for SDN controllers. To see how SODA performs compared to an existing solution, we conduct the same experiments on a NOX controller [12], which is one of the most lightweight SDN controllers and is suitable for IoT environments. In the NOX controller, we run a simple forwarding application to deliver given requests that follow the SODA protocol to IoT devices.

Figure 8 shows the throughput variations of SODA and NOX. In our experiments, we increase the number of IoT devices connected to SODA and NOX from 32 to 1,024 devices. From the results, we observe that the total number of requests that each system can handle is almost identical no matter how many devices are connected. However, we see that the number of requests SODA can handle (164.9K on average) is almost twice than those of NOX (79.0K on average). For example, with 1,024 devices, while NOX can handle 77 requests per device, SODAprocesses at least 161 requests per device simultaneously.

Latency Measurement: Here, we also measure the processing latencies of SODA and NOX. In these measurements, the latency means the difference between the time when the data plane sends a controller a request and the time when the data plane receives a response from the controller. Also, all devices in each case separately send requests to either SODA or NOX in parallel. Since the policy enforcement happens before data transactions, we thus assume that required policies are already enforced into the data plane.

Figure 9 shows the processing latencies of SODA and NOX. From the results, we see that the latencies of SODA (0.8 ms on average) are 34.4% lower than those of NOX (1.22 ms on average) even though SODA requires some time to look up enforced security policies, and the reason that SODA shows better performance than NOX is mostly because of our highly parallelized event-driven model.

Based on those measurement results, we see that SODA can achieve much higher performance than an existing solution (i.e., NOX) while SODA provides powerful security policy mechanisms. Furthermore, these results indicate how SODA is practical in a real IoT environment.

7. Real-World Evaluation

Now, we evaluate the effectiveness and practicality of SODA by demonstrating network attacks discussed in Section 2, and describe how SODA can mitigate these attacks

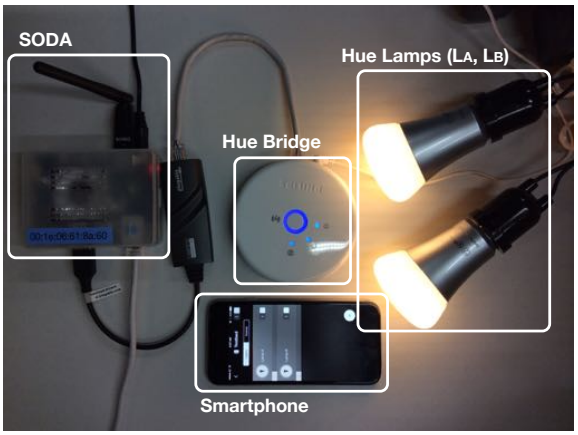


Figure 10: Test environment with SODA and real IoT devices

through security applications running on our system.

7.1. Test Environment

For the demonstrations of attack scenarios, we have installed the prototype of our system in a real IoT environment, as shown in Figure 10, where two client devices (a laptop and a smartphone) and a set of Philips Hue devices that include a Hue bridge and two Hue lamps (denoted as L_A and L_B) have been deployed. Note that We have used unmodified version of binary images for each device to show its practicality.

As we assumed in Section 2, the lamp L_A is under the control of the laptop while the lamp L_B and the bridge are under the control of the smartphone. The only difference between this test environment and the environment used in Section 2 is that we have replaced the legacy access point (AP) to our secure IoT gateway.

The Hue bridge communicates with a client device (i.e., the smartphone) using JSON over HTTP. To translate messages between the bridge and the client, we have deployed a protocol parser that has been implemented in the control plane as a SODA application by using SODA APIs, motivated by previous work [19, 25]. In addition, since the Odroid board that we have used for our prototype does not support a zigbee interface, we have linked a Philips Hue bridge with our system and used it as an extended zigbee hardware module to communicate with Hue lamps.

7.2. Security Policy Management

SODA maintains all security policies that are specified and enforced by each user; thus, it can understand the security requirements of users and provide conflict-free and bespoke security services for them. To show the feasibility of our policy management scheme, we describe how SODA works in a real environment with low overhead through an *automatic policy enforcer* application running on SODA.

7.2.1. User-defined Access Control

Without SODA, the laptop can control the lamp L_B, which is owned by the smartphone using the open Hue APIs because there is no access control in an unsecured IoT network. To block this undesired device access, the smartphone owner can enforce the following security policy:

$$\text{Laptop} \rightarrow L_B : \text{State, Read\&Write} \mid \text{Deny} \qquad (P_1)$$

SODA then examines any conflict between the policy P_1 and resident policies. Since the network has no installed policies at the beginning, the policy P_1 is accepted and installed into the network. A message from the laptop to turn off the lamp L_B is dropped because it violates the policy P_1, and SODA installs a drop rule for such access into the data plane. Now, we assume that the laptop tries to enforce the policy P_2 which allows the laptop to control the lamp L_B:

$$\text{Laptop} \rightarrow L_B : \text{State, Write} \mid \text{Allow} \qquad (P_2)$$

After this policy is submitted to SODA, the policy management scheme of SODA detects a conflict with policy P_1 and tries to resolve the conflict. According to our resolution mechanism, policy P_2 is rejected because it cannot be fragmented (we assume all user policies have the same priority).

In reality, it is challenging for an IoT user to specify and enforce proper security policies for IoT devices due to the dynamic connectivity of IoT devices. If a user wants to control any accesses from/to IoT devices, the user must know whether the IoT devices have already joined into the network. Also, when new IoT devices join into the network, there is no policy for them; thus, it is important to recognize when the devices have exactly joined to the network. For instance, let us assume that a new user joins the network with a tablet device. As the network has no related policies for the tablet, the new user can enforce policies to access the laptop or the smartphone until this new device connection is noticed by existing users and appropriate security policies are enforced. To overcome such challenges coming from manual policy enforcement and to achieve a reactive security-enforcing environment, we have implemented an *automatic policy enforcer*, which automatically create and enforce semantic security policies for newly joined devices.

7.2.2. Automatic Policy Enforcement

The automatic policy enforcer has three major functions: i) receiving semantic policies from users, ii) detecting new devices for reactive security enforcement, and iii) deriving security policies for new devices from the reference semantics without distorting user intentions. Here, semantic policies mean a set of simple statements, which helps a user to create desired security policies without the knowledge of either a network or its security. Since SODA manages security policies only without semantic information, this application maintains semantic policies in its internal policy table, and then transform them into finger-grained policies when a new device joins into a network.

To detect the arrival of a new device, the application listens for device-related events and reads authentication data through a state-reading API to identify the owner of the device. According to this information, the application groups devices that have the same (or related) semantics to facilitate a semantic-level management. After grouping devices that are connected to the network based on their semantics,

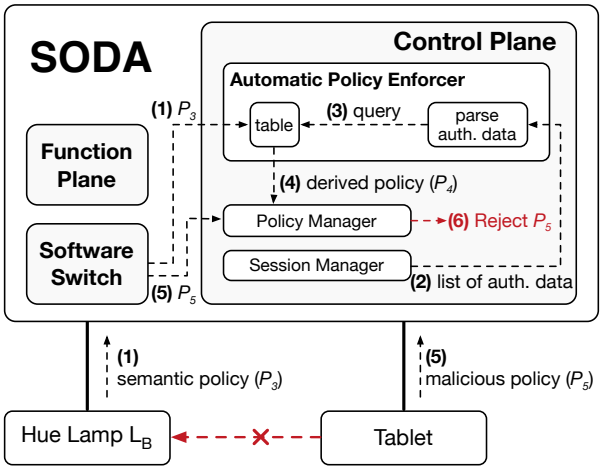


Figure 11: Operation scenario for automatic policy enforcer

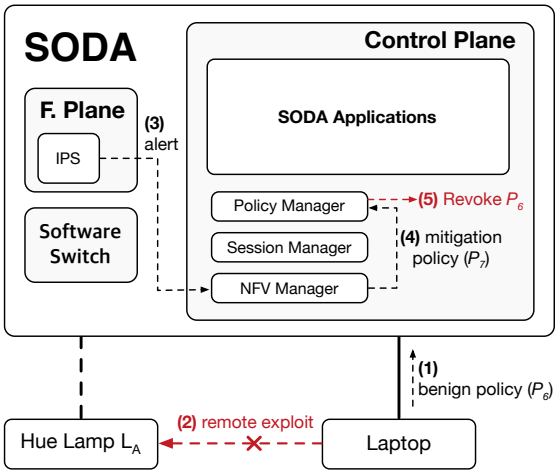


Figure 12: Operation scenario for remote exploit detection

the application queries its internal policy table, which contains a list of semantic policies enforced by users. If the table contains a matched semantic policy, the application derives fine-grained policies from it by replacing a semantic-level identifier with a low-level identifier. Then, the application submits the fine-grained policies to the policy management of the core module through a policy-enforcing API. In the end, the network reactively provides user-defined security without user intervention.

Operational Scenario: Figure 11 illustrates the workflow of the automatic policy enforcer. We assume that the owner of the lamp L_B installs the policy P_3, which defines that no other component apart from the owner's devices can access data from the bridge, instead of the policy P_1:

$$\text{NotMine} \rightarrow L_B\text{:State, Read\&Write} \mid \text{Deny} \qquad (P_3)$$

When the application receives this policy, it derives fine-grained policies by reading information from connected devices in the network. Since the laptop is not owned by the owner of the bridge, the application automatically derives and installs the policy P_1 from the semantic policy. After that, we connect a new tablet device into the network as a new user. Once the connection is established, the automatic policy enforcer receives a device-registration event and identifies the owner of the tablet. Based on this, the application automatically constructs and installs a corresponding fine-grained policy P_4 by referring to the semantic policy P_3:

$$\text{Tablet} \rightarrow L_B\text{:State, Read\&Write} \mid \text{Deny} \qquad (P_4)$$

To confirm that the application generates the appropriate policy and SODA correctly detects a policy conflict, we consider a case in which the tablet tries to control the lamp L_B. In this case, the tablet can create and submit a policy:

$$\text{Tablet} \rightarrow L_B\text{:State, Write} \mid \text{Allow} \qquad (P_5)$$

When policy P_5 is transmitted to SODA, its policy management scheme detects a conflict with policy P_4 because both policies have overlapping match fields but different actions. By following a similar resolution step as the previous scenario, policy P_5 is rejected because it cannot be fragmented.

7.3. Network Attack Mitigation

The powerful network connectivity of IoT environments exposes IoT devices to existing network attacks. Unfortunately, most IoT devices have no defense mechanisms for such attacks. To address both internal network attacks (between user devices) and external network attacks, we utilize a programmable interface to implement a range of security applications. and this capability is integrated into the function plane of SODA where virtualized security functions are offloaded. Here, we describe how SODA can detect and mitigate network attacks through two example functions running on SODA: an intrusion prevention system on the function plane and an anomaly detector application on the control plane of SODA.

7.3.1. Remote Exploit Detection

Some attackers analyze the binary image of a device to find exploitable vulnerabilities such as buffer overflow, and try to acquire higher privileges of a target IoT device. To protect IoT devices from such attempts, we have implemented an *intrusion prevention system* (IPS), which inspects the contents of all network flows in a network to find known attack patterns, on the function plane of SODA. When the IPS detects a known attack pattern, it reports the details of the detected attack to the control plane of SODA. Based on this information, the core module specifies mitigation policies and installs them with a high priority (i.e., the security role) not to be affected by the policies with a low priority (i.e., the user role) due to any policy conflicts.

Operational Scenario: There is a known vulnerability that allows an attacker to overwrite the binary image of the Hue bridge with a malicious one; thus, an attacker exploits this vulnerability by modifying the name of a connected Hue lamp to execute an arbitrary code. Then, the bridge executes the injected code to download and update a malicious image

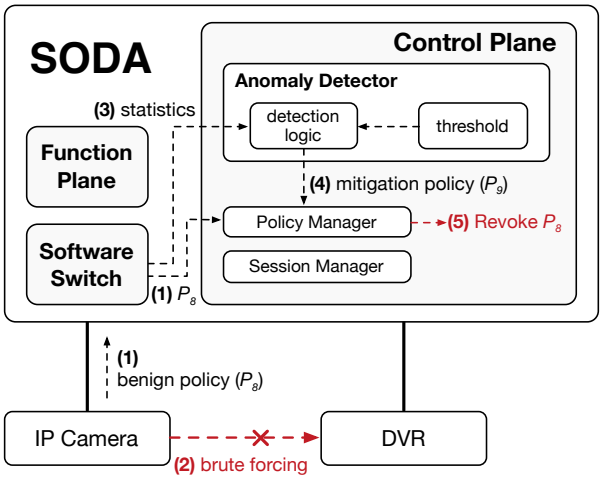


Figure 13: Operation scenario for botnet mitigation

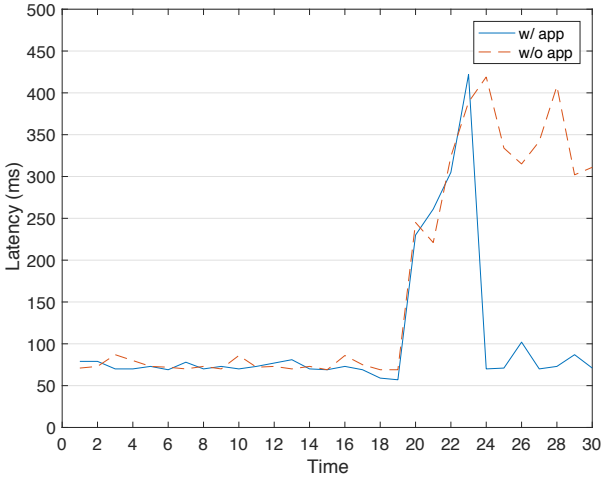


Figure 14: Inter-device latency variations during a DoS attack

when any user tries to read the modified lamp name stored in the bridge. Figure 12 shows the operational scenario for detecting and mitigating a remote exploit using the IPS.

We assume that the owner of the laptop exploits the vulnerability of the Hue bridge, and the IPS has an appropriate detection mechanism. To send an exploit message to the lamp L_A, the attacker enforces the following policy:

$$\text{Laptop} \rightarrow L_A : \text{State, Read\&Write} \mid \text{Allow} \qquad (P_6)$$

Since there is no conflict with previous policies, the policy P_6 is successfully installed to SODA. When the attacker sends an exploit message to the lamp L_A, SODA first forwards it to the IPS in the function plane before delivering to the lamp L_A. As the IPS detects the attack pattern of the vulnerability from the incoming message, SODA drops the message immediately while reporting the detected attack to the core module. In the end, SODA enforces a mitigation policy that blocks the attack from the laptop:

$$\text{Laptop} \rightarrow L_A : \text{State, Read\&Write} \mid \text{Deny} \qquad (P_7)$$

After submitting policy P_7, SODA revokes policy P_6. This is because policy P_6 and P_7 are conflict with each other while policy P_7 has a higher priority (security-role) than policy P_6 (user-role).

7.3.2. Botnet Mitigation

IoT malware is another crucial threat in an IoT environment. As IoT devices have at least sufficient computing power to craft and send packets, albeit limited, attackers can exploit this feature to send attack packets from IoT devices to other victims, including traditional network devices. To see the behavior of IoT bots, we have deployed an IoT device infected by Mirai [3] in our IoT environment. A Mirai bot first scans an internal network to find potential victims that host a telnet or SSH service, and then performs a brute-force attack to find its username and password. This information is reported to a remote server, and the server sends device-specific malware to victims.

To mitigate such internal botnet propagation, we have implemented an *anomaly detector* which monitors a network to detect abnormal telnet and SSH connection trials that send SYN packets over a threshold in a short period (e.g., every 5 seconds). Then, if any abnormal trials are detected, the application enforces security policies to block the trials.

Operational Scenario: In this example, we deploy two devices (an IP Camera infected by Mirai and a vulnerable DVR device) in a SODA network. Figure 13 depicts the operational scenario for detecting and mitigating Mirai propagation using the anomaly detector application.

Since there is no security policy for blocking the accesses to internal devices, an attacker can successfully enforce security policies for allowing a telnet communication with connected devices. As an example, the infected IP Camera can scan the DVR after enforcing the following policy:

$$\text{IPCamera} \rightarrow \text{DVR} : \text{Telnet, Write} \mid \text{Allow} \qquad (P_8)$$

Using this policy, the IP Camera can execute a brute-force attack against the DVR to acquire its telnet shell. Since these messages do not violate any user-defined security policy, the network delivers all messages to the DVR. However, after the IP Camera begins an attack against the DVR, the anomaly detector identifies abnormal telnet accesses and thus enforces a mitigation policy that denies access from the IP Camera to the DVR using the following policy:

$$\text{IPCamera} \rightarrow \text{DVR} : \text{Telnet, Read} \mid \text{Deny} \qquad (P_9)$$

Then, this policy revokes policy P_8 because policy P_8 and P_9 are conflict with each other, and policy P_9 has a higher priority than policy P_8. As a result, SODA can successfully mitigate the Mirai propagation by detecting and blocking the brute-force attack using the anomaly detector.

Figure 14 shows the latency variations between the smartphone and the bridge by sending simple data and receiving a response. At 20 seconds, the laptop starts a DoS attack against the Hue bridge. Without the anomaly detector, the message delivery latency increases almost 5-fold after the

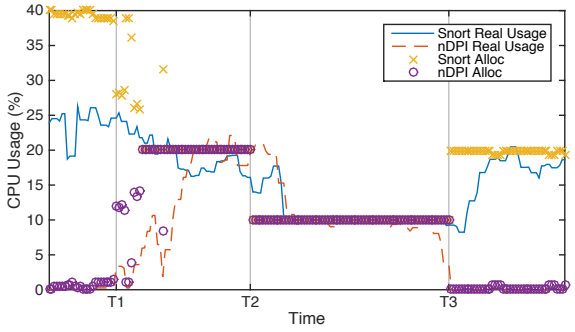


Figure 15: Resource management in the function plane

laptop executes a DoS attack on the Hue bridge. However, as soon as the application runs on SODA, it detects and blocks the attack using mitigation policies similar to policy P_9 and thus the quality-of-service experience of the smartphone can be maintained at approximately 77ms.

7.4. Function Plane Resource Management

As various network services are running on our secure IoT gateway, SODA controls the resource of each function to guarantee that the network can provide full security functionalities without any disruption. Also, SODA manages available resources to maximize the overall resource utilization of the function plane.

To evaluate the resource management of SODA, we have installed two security functions in the function plane. One is Snort [43], which is a well-known open-source intrusion detection system (IDS) [1], and the other is nDPI [8], which is an open-source deep packet inspection (DPI) system. In our test scenario, nDPI is only activated when an attack alert is triggered by the IDS; otherwise, it remains idle. We assume that those functions have the same weight and priority, and our gateway has 40% of available (idle) CPU resources.

As shown in Figure 15, at the beginning, all idle resources are allocated to Snort until `T1` since there is no detected attack. At `T1`, Snort raises an attack alert, and nDPI is activated by taking the half of the idle resources (i.e., 20%). Now, let us assume that SODA allocates the half of the remaining idle resources to some other functions at `T2`. Then, due to this resource restriction, both functions have halved resources (i.e., 10%). Once the detected attack is blocked and there is no additional attack, nDPI is deactivated. In the end, the resource allocated to nDPI is reallocated to Snort (at `T3`).

8. Related Work

IoT Framework: In the IoT field, there have been some research works that have designed frameworks for IoT security. For example, Roesner *et al.* have focused on privacy issues caused by untrusted IoT entities and proposed a framework for the access control of IoT data with certificate-based policy authenticity [33]. Sicari *et al.* have proposed a risk assessment framework that evaluates a risk in an IoT system according to the life cycle of IoT data [41]. While these studies have addressed some security issues in an IoT environment, their goals differ from our work.

Some others have addressed IoT security challenges by employing the concept of SDN in their systems. Yu *et al.* [46] have analyzed IoT security challenges and envisioned a high-level architecture of a software-defined policy enforcement framework. Similarly, Lorenz *et al.* [17] and Salman *et al.* [34] have employed the concept of SDN to achieve security policy enforcement for IoT. Here, since SODA and their solutions have adopted the concept of SDN, their high-level architecture could be similar to each other. However, we note that SODA provides more detailed and realistic security management methods than their systems. Moreover, SODA provides a lightweight NFV environment for IoT security services, which has not been done by others, and we have also demonstrated that our solution can address security issues in the real IoT environment effectively and efficiently.

IoT Gateway: Zhu *et al.* have proposed an IoT gateway that supports the conversion of different sensor network protocols to enable the seamless integration of IoT devices [48]. While SODA does not support other protocols (e.g., Zigbee [49] and oneM2M [22]) yet, we plan to extend SODA to support those well-known protocols. Sicari *et al.* have proposed a cross-domain IoT middleware that filters IoT data delivered to IoT users and services according to their security and data quality levels [38, 39]. On the other hand, SODA specifically concentrates on the policy conflict issues among security policies. Thus, their goals differ from our work.

With regard to communications between IoT users and an IoT gateway, Rizzardi *et al.* have introduced a secure publish/subscribe system (AUPS) for multiple IoT users [32]. On the other hand, the current prototype of SODA only allows a single administrator to manage SODA for policy enforcement and security service deployment, but AUPS could be adopted into SODA for multi-user support.

In industry, Cisco [7] and Freescale [18] have launched integrated service routers designed for IoT, and supported integrated communications and security features (e.g., Firewall) for IoT devices. Unfortunately, their IoT gateways do not satisfy device-side access control requirements. To complement this, SODA can be used to fill such gap and achieve a novel device-side access control for an IoT environment.

IoT Security: There have been some researches focusing on IoT security issues. Babar *et al.* [6] and Riahi *et al.* [31] have analyzed security and privacy issues and proposed security models based on IoT objects and their interactions. Zhao *et al.* have presented security requirements (e.g., authentication and access control) for an IoT system [47]. More specifically, Fernandes *et al.* have conducted security evaluations against a commercial IoT gateway [35] and analyzed its vulnerabilities [11]. Unfortunately, none of them have considered advanced security threats (e.g., conflicts among security policies) or user-driven security services. In contrast, SODA provides a comprehensive solution to address a range of IoT security issues, and demonstrates its functionalities based on our practical implementation and evaluation.

[1] https://www.snort.org

9. Limitations and Future Work

As do other research proposals, SODA also has some limitations. As previously noted, SODA has not considered some security features (e.g., lightweight encryption and DoS attacks against a centralized IoT gateway) for an IoT environment in this prototype implementation, as they have already been presented by several previous studies [29, 42]. However, we believe that such security features can be easily integrated into SODA as additional security services using our flexible and extensible NFV functionalities.

In terms of our future work, although we have focused on a centralized platform (i.e., we handle all IoT-related network traffic with a single gateway) in this paper, it could be hard to manage the security policies for all IoT devices that are geographically distributed. Hence, a distributed platform that covers those geographically distributed IoT devices would be required while it should consider complex security policy management for handling policy synchronization [40] and handover issues as well. Along with this, a distributed policy management including the conflict resolution among the policies located in different IoT gateways would be needed.

10. Conclusion

The security of the Internet of Things (IoT) has been challenged due to the insufficient capabilities of IoT devices and security policies enforced from multiple sources to the devices. As a result, while IoT devices unwillingly provide limited security functionalities, security policies could be conflicted with each other due to different intents among IoT users. To solve those challenges, we have introduced SODA, which leverages SDN and NFV techniques to offer a centralized security management for an IoT environment. SODA has been designed to provide a user-defined access control scheme with policy conflict detection and resolution and practical IoT security services with the assistance of security service offloading for resource-limited IoT devices. We have demonstrated its effectiveness and practicality with several attack scenarios in a real IoT environment. Our performance results also show that SODA can support a real IoT environment with satisfactory performance. While this work has focused on a home IoT environment, we believe that SODA can be easily extended to other IoT environments (e.g., office and enterprise environments) as well.

Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00254, SDN security technology development).

References

[1] Abadi, M., 2007. Access Control in a Core Calculus of Dependency. Electronic Notes in Theoretical Computer Science 172, 5–31.

[2] Abadi, M., Burrows, M., Lampson, B., Plotkin, G., 1991. A Calculus for Access Control in Distributed Systems, in: Proceedings of Annual International Cryptology Conference (AICC '91), Springer. pp. 1–23.

[3] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M., et al., 2017. Understanding the Mirai Botnet, in: Proceedings of USENIX Security Symposium (USENIX Security '17), pp. 1093–1110.

[4] Apple Inc., 2019a. Apple HomeKit. http://www.apple.com/ios/homekit; Last accessed: 08/01/2019.

[5] Apple Inc., 2019b. Apple Watch. http://www.apple.com/watch; Last accessed: 08/01/2019.

[6] Babar, S., Mahalle, P., Stango, A., Prasad, N., Prasad, R., 2010. Proposed Security Model and Threat Taxonomy for the Internet of Things (IoT), in: Proceedings of International Conference on Network Security and Applications (ICNSA '10), Springer. pp. 420–429.

[7] Cisco Systems Inc., 2019. Cisco 910 Industrial Router. https://www.cisco.com/c/en/us/support/routers/910-industrial-router/model.html; Last accessed: 08/01/2019.

[8] Deri, L., Martinelli, M., Bujlow, T., Cardigliano, A., 2014. nDPI: Open-source High-speed Deep Packet Inspection, in: Proceedings of International Wireless Communications and Mobile Computing Conference (IWCMC '14), IEEE. pp. 617–622.

[9] Dhanjani, N., 2015. Abusing the Internet of Things: Blackouts, Freakouts, and Stakeouts. "O'Reilly Media, Inc.".

[10] ETSI, 2012. Network Functions Virtualisation (NFV). White Paper.

[11] Fernandes, E., Jung, J., Prakash, A., 2016. Security Analysis of Emerging Smart Home Applications, in: Proceedings of IEEE Symposium on Security and Privacy (SP '16), IEEE. pp. 636–654.

[12] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., Shenker, S., 2008. NOX: Towards an Operating System for Networks. Proceedings of ACM SIGCOMM Computer Communication Review (CCR '08) 38, 105–110.

[13] HardKernel, 2019. Odroid XU3. https://www.hardkernel.com/ko/shop/odroid-xu3; Last accessed: 08/01/2019.

[14] Lantz, B., Heller, B., McKeown, N., 2010. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks, in: Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets '10), ACM. p. 19.

[15] Linux Foundation, 2016. Open vSwitch. http://openvswitch.org/; Last accessed: 08/01/2019.

[16] Linux Foundation, 2019. IoTivity project. https://www.iotivity.org; Last accessed: 08/01/2019.

[17] Lorenz, C., Hock, D., Scherer, J., Durner, R., Kellerer, W., Gebert, S., Gray, N., Zinner, T., Tran-Gia, P., 2017. An SDN/NFV-Enabled Enterprise Network Architecture Offering Fine-Grained Security Policy Enforcement. IEEE communications magazine 55, 217–223.

[18] Maguire, J., 2014. Internet of Things (IoT) Service Delivery using NFV/SDN. Freescale Technology Forum. FTF-NET-F0160.

[19] Mekky, H., Hao, F., Mukherjee, S., Zhang, Z.L., Lakshman, T., 2014. Application-aware Data Plane Processing in SDN, in: Proceedings of the Workshop on Hot topics in Software Defined Networking (HotSDN '14), ACM. pp. 13–18.

[20] Nam, J., Jo, H., Kim, Y., Porras, P., Yegneswaran, V., Shin, S., 2019. Operator-Defined Reconfigurable Network OS for Software-Defined Networks. IEEE/ACM Transactions on Networking .

[21] Narm, J., Jot, H., Kim, Y., Porras, P., Yegneswaran, V., Shin, S., 2018. Barista: an Event-centric NOS Composition Framework for Software-Defined Networks, in: Proceedings of IEEE Conference on Computer Communications (INFOCOM '18), IEEE. pp. 980–988.

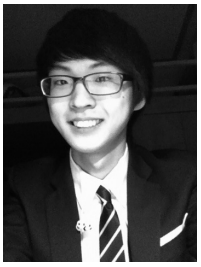
[22] oneM2M, 2017. Standards for M2M and the Internet of Things. http://www.onem2m.org/; Last accessed: 08/01/2019.

[23] Open Connectivity Foundation, 2019. AllJoyn project. https://openconnectivity.org; Last accessed: 08/01/2019.

[24] Open Networking Foundation, 2012. Software-defined networking: The new norm for networks. White Paper.

[25] Park, T., Kim, Y., Shin, S., 2016. UNISAFE: A Union of Security Actions for Software Switches, in: Proceedings of the ACM Interna-

tional Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN&NFV '16), ACM. pp. 13–18.
[26] Philips Co., 2019. Philips Hue. http://www2.meethue.com/en-us/; Last accessed: 08/01/2019.
[27] Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., Gu, G., 2012. A Security Enforcement Kernel for OpenFlow Networks, in: Proceedings of the workshop on Hot topics in Software Defined Networks (HotSDN '12), ACM. pp. 121–126.
[28] Porras, P.A., Cheung, S., Fong, M.W., Skinner, K., Yegneswaran, V., 2015. Securing the Software Defined Network Control Layer, in: Proceedings of (NDSS '15).
[29] Prasetyo, K.N., Purwanto, Y., Darlis, D., 2014. An implementation of data encryption for Internet of Things using blowfish algorithm on FPGA, in: Proceedings of International Conference on Information and Communication Technology (ICoICT '14), IEEE. pp. 75–79.
[30] Raspberry Pi Foundation, 2019. Raspberry Pi. https://www.raspberrypi.org; Last accessed: 08/01/2019.
[31] Riahi, A., Challal, Y., Natalizio, E., Chtourou, Z., Bouabdallah, A., 2013. A systemic approach for IoT security, in: Proceedings of IEEE International Conference on Distributed Computing in Sensor Systems (ICDCSS '13), IEEE. pp. 351–355.
[32] Rizzardi, A., Sicari, S., Miorandi, D., Coen-Porisini, A., 2016. AUPS: an open source AUthenticated publish/subscribe system for the internet of things. Information Systems 62, 29–41.
[33] Roesner, F., Molnar, D., Moshchuk, A., Kohno, T., Wang, H.J., 2014. World-driven access control for continuous sensing, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '14), ACM. pp. 1169–1181.
[34] Salman, O., Elhajj, I., Chehab, A., Kayssi, A., 2017. Software Defined IoT Security Framework, in: Proceedings of International Conference on Software Defined Systems (SDS '17), IEEE. pp. 75–80.
[35] Samsung Inc., 2019. Samsung SmartThings. https://www.smartthings.com/; Last accessed: 08/01/2019.
[36] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M.K., Shi, G., 2012. Design and Implementation of a Consolidated Middlebox Architecture, in: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '12), pp. 323–336.
[37] Sherwood, R., Kok-Kiong, Y., 2019. Cbench - a benchmarking tool for SDN controllers. https://github.com/mininet/oflops/tree/master/cbench; Last accessed: 08/01/2019.
[38] Sicari, S., Rizzardi, A., Miorandi, D., Cappiello, C., Coen-Porisini, A., 2016a. A secure and quality-aware prototypical architecture for the Internet of Things. Information Systems 58, 43–55.
[39] Sicari, S., Rizzardi, A., Miorandi, D., Cappiello, C., Coen-Porisini, A., 2016b. Security Policy Enforcement for Networked Smart Objects. Computer Networks 108, 133–147.
[40] Sicari, S., Rizzardi, A., Miorandi, D., Coen-Porisini, A., 2017. Dynamic Policies in Internet of Things: Enforcement and Synchronization. IEEE Internet of Things Journal 4, 2228–2238.
[41] Sicari, S., Rizzardi, A., Miorandi, D., Coen-Porisini, A., 2018a. A risk assessment methodology for the Internet of Things. Computer Communications 129, 67–79.
[42] Sicari, S., Rizzardi, A., Miorandi, D., Coen-Porisini, A., 2018b. REATO: REActing TO Denial of Service attacks in the Internet of Things. Computer Networks 137, 37–48.
[43] Snort, 2019. Snort. https://www.snort.org/; Last accessed: 08/01/2019.
[44] The GNOME Project, 2014. GLib. https://developer.gnome.org/glib/stable/; Last accessed: 08/01/2019.
[45] Thread Group, 2019. Thread. http://threadgroup.org; Last accessed: 08/01/2019.
[46] Yu, T., Sekar, V., Seshan, S., Agarwal, Y., Xu, C., 2015. Handling a trillion (unfixable) flaws on a billion devices: Rethinking Network Security for the Internet-of-Things, in: Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets '15), ACM. p. 5.
[47] Zhao, K., Ge, L., 2013. A survey on the Internet of Things Security, in: Proceedings of International Conference on Computational Intelligence and Security (ICCIS '13), IEEE. pp. 663–667.
[48] Zhu, Q., Wang, R., Chen, Q., Liu, Y., Qin, W., 2010. IoT Gateway: Bridging Wireless Sensor Networks into Internet of Things, in: Proceedings of IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (ICEUC '10), IEEE. pp. 347–352.
[49] ZigBee Alliance, 2018. ZigBee. http://www.zigbee.org/; Last accessed: 08/01/2019.

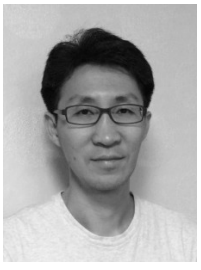


Yeonkeun Kim is a Ph.D. student in Graduate School of Information Security at KAIST. He received his B.S. degree in Computer Science Engineering at Ulsan National Institute of Science and Technology (UNIST) in Korea. He received his M.S. degree in Information Security from KAIST. His research interests include network security issues of IoT and embedding systems.

Jaehyun Nam is a Ph.D. student in Graduate School of Information Security at KAIST. He received his B.S. degree in Computer Science and Engineering from Sogang University in Korea. He received his M.S. degree in Information Security from KAIST. His research interests focus on networked and distributed computing systems. He is especially interested in performance and security issues from software-defined networking (SDN) and network function virtualization (NFV).

Taejune Park is currently pursuing his Ph.D. degree in School of Computing at KAIST, Republic of Korea, from September 2015. He received his B.S. degree in Computer Engineering at Korea Maritime and Ocean University, Republic of Korea, in August 2013, and his M.S. degree in Information Security at KAIST, Republic of Korea, in August 2015. His research interests focus on the security issues on SDN/NFV environments and data-planes.

Dr. Sandra Scott-Hayward, CEng CISSP CEH, is a Lecturer (Assistant Professor) in Network Security at QueenâĂŹs University Belfast. In the Centre for Secure Information Technologies at QUB, Sandra leads research and development of network security architectures and security functions for SDN and NFV. She has presented her research globally and received Outstanding Technical Contributor and Outstanding Leadership awards from the Open Networking Foundation (ONF) in 2015 and 2016, respectively.



Seungwon Shin is an associate professor in the School of Electrical Engineering at KAIST. He received his Ph.D. degree in Computer Engineering from the Electrical and Computer Engineering Department, Texas A&M University, and his M .S degree and B.S degree from KAIST, both in Electrical and Computer Engineering. He is currently a Research Associate of Open Networking Foundation (ONF), and a member of security working group at ONF. His research interests span the areas of SDN security, IoT security, and Botnet analysis/detection.