Generating Stateful Policies for IoT Device Security with Cross-Device Sensors

Zorigtbaatar Chuluundorj, Shuwen Liu and Craig A. Shue
Worcester Polytechnic Institute
{zchuluundorj, sliu9, cshue}@wpi.edu

***Abstract*—The security of Internet-of-Things (IoT) devices in the residential environment is important due to their widespread presence in homes and their sensing and actuation capabilities. However, securing IoT devices is challenging due to their varied designs, deployment longevity, multiple manufacturers, and potentially limited availability of long-term firmware updates. Attackers have exploited this complexity by specifically targeting IoT devices, with some recent high-profile cases affecting millions of devices.**

In this work, we explore access control mechanisms that tightly constrain access to devices at the residential router, with the goal of precluding access that is inconsistent with legitimate users' goals. Since many residential IoT devices are controlled via applications on smartphones, we combine application sensors on phones with sensors at residential routers to analyze workflows. We construct stateful filters at residential routers that can require user actions within a registered smartphone to enable network access to an IoT device. In doing so, we constrain network packets only to those that are consistent with the user's actions. In our experiments, we successfully identified 100% of malicious traffic while correctly allowing more than 98% of legitimate network traffic. The approach works across device types and manufacturers with straightforward API and state machine construction for each new device workflow.

I. Introduction

Internet-of-Things (IoT) devices have entered the mass-market adoption phase in the residential environment, particularly consumer electronics such as smart power outlets, light bulbs, fans, speakers, and media-streaming devices. Unfortunately, IoT devices are beset by security challenges. These devices are often embedded into a home for years or decades before being replaced. End users may not properly configure them for security or maintain them. Some device manufacturers offer firmware updates to keep their products secure; however, such software availability may vary by device and manufacturer, and the updates may not be applied consistently by end-users. As a result, IoT devices may have long-standing vulnerabilities and pose an attractive target for malicious actors [28], [26].

Past incidents include the 2016 Mirai botnet [4], a smart deadbolt vulnerability to unlock doors remotely [23], and a monitoring device that allowed malicious actors to spy on individuals and obtain passwords for WiFi networks [23]. With surveys indicating that 2 million IoT devices are vulnerable to complete takeover [22], such incidents may continue in the future.

Security concerns surrounding IoT devices have led to significant prior work, including in the device classification [27], [21], [20], mobile application [14], [6], and vulnerability analysis spaces [32], [34]. These studies have the same constraint: they attempt to secure IoT devices by looking only at the IoT device itself. Since end-users typically control IoT devices remotely via smartphone apps, we believe these application interactions may provide data for increasing IoT device security.

When controlling IoT devices, end-users typically express a command via a smartphone application. These commands may result in multiple network packet transmissions to fulfill the request. By organizing these device interactions into state machines, we can model these high-level actions and only allow recognized commands that are consistent with an end-user's actions on a phone. Such an approach would disrupt malformed packets attempting to exploit a vulnerability, prevent usage by unauthorized users with stolen account credentials, and prevent malware on the phone from taking covert IoT actions. Further, these controls would allow network controllers to enforce higher-level policy (e.g., "allow the power to toggle no faster than once every 3 minutes").

In this work, we ask two research questions: *To what extent can we link end-user interactions within an IoT device's smartphone application with the resulting network flows to that IoT device? To what extent can we leverage those user interactions in filtering network traffic for IoT devices?* In exploring these questions, we add sensors to both smartphone apps and residential routers to monitor and link behaviors.

This work makes the following contributions:

- **Creation of a cross-device sensing prototype system:** We use the Android `AccessibilityServices` to monitor smartphone user interface (UI) events. At the residential router, we use software-defined networking (SDN) to intercept and inspect IoT device traffic.
- **Fusion of user actions with network traffic:** At an SDN controller, we link the sensor data from the smartphone and SDN router. We relate UI interactions with the following network activity to establish causal links. We then create network protocol state machines associated with each UI behavior to allow the ensuing traffic.
- **Evaluation of the prototype's efficacy and performance:** We explore the ability of the prototype to distinguish IoT traffic related to UI activity from traffic that is not related. After generating policy from training data, we find our technique can correctly classify over 98% of legitimate traffic and can identify 100% of malicious

traffic. Our performance measurements of the system's impact on the end-to-end IoT behaviors reveal a modest impact on latency and computational overheads.

The paper is structured as follows. In Section II, we present related work. We describe our approach in Section III with implementation details in Section IV. We describe the results of our empirical study in Section V and conclude in Section VI.

II. Related Work

Prior work on defending IoT devices from malicious actors has approached the problem from multiple directions, often focusing on a single aspect of IoT activity, such as the application user interface, the inner workings of the application, or the network activity.

Prior work has explored applying software-defined networking (SDN) techniques to protect residential IoT devices. Taylor et al. [31] considered the feasibility of using cloud-based SDN controllers for residential networks. Liu et al. [17] later used SDN and a cloud-based service to implement two-factor authentication for IoT devices. Sivanathan et al. [29] proposed a flow-based network defense of IoT devices with SDN. Yu et al. [36] proposed using SDN and a cloud-based service to store malicious attack signatures to protect IoT devices. Sivaraman et al. [30] proposed using SDN to implement dynamic security rules that vary based on context, such as time-of-day or occupancy of the house. In contrast to these SDN efforts, which focus solely on network traffic in the home, our work links network behavior with the user actions that caused the traffic.

Both static and dynamic analysis have been applied to the inner workings of smartphone applications to detect anomalies, with some utilizing UI analysis to drive the application. Lindorfer et al. [16], Carter et al. [5], and Blasing et al. [3] proposed Andrubis, CuriousDroid, and AASandbox, respectively. The proposed tools are fully automated, using the Monkey [2] tool, and use both static and dynamic analysis to examine Android at the system level. These hybrid approaches identified potentially malicious behavior such as dynamic code landing, SMS-related code, and network activity indicating potentially malicious behavior. Yang et al. [35] used static analysis to locate conditional statements that lead to security-sensitive behaviors, and they used Support Vector Machine to classify applications as malicious or benign. Wong et al. [33] developed a tool, IntelliDroid, to assist dynamic analysis by generating configured inputs into applications. However, these prior efforts focused only on phone UI and did not draw clear causation between UI and IoT network traffic. Further, these approaches considered UI as a complementary component that enables the dynamic analysis, not as a view into the inner workings and network activity of an application.

The body of work utilizing UI for security purposes has focused on correlating UI with network activity to detect malicious network behavior. SUPOR used static analysis to identify UI elements that take sensitive user inputs as potential privacy or security risks [13]. Homonit [37] and IoTGaze [12] compared IoT network activity to expected behavior indicated by their source code, UI, and application descriptions in order to detect potential system anomalies and hidden vulnerabilities. Fu et al. [10] proposed a similar approach, but only considered foreground UI text; they used Natural Language Processing (NLP) to determine if network activity was justified. Chen et al. [6] proposed IoTFuzzer, which detects IoT memory corruption vulnerabilities (without requiring access to the firmware) by hooking into, and mutating, UI input fields. Gianazza et al. [11] used recorded UI interactions from malicious applications to determine if similar UI sequences yield malicious behavior in other Android applications. Other work related to the Android UI has focused on profiling user actions in an Android application by observing encrypted traffic generated by the applications with the use of machine learning [7]. Their work used a coarse-grained analysis of the UI to correlate it with the internal workings of an application and network activity. In contrast, our approach focuses on linking interactions with specific UI widgets to specific protocol commands and network patterns at the IoT device to enable cross-device, fine-grained analysis.

Some work on IoT device security has focused on limiting network activity based on device types. Ngyuan et al. [21] built device-type-specific communication profiles for detecting compromises in IoT devices. Similarly, Miettinen et al. [20] proposed a system that automatically identifies the device type of newly connecting devices as they join an IoT network, enabling enforcement of rules for limiting the communications of vulnerable devices. The Edgesec tool [27] categorizes devices based on their capabilities and monitors network activity between IoT devices and the outside network, and offloads the security responsibilities to the edge layer. Jia et al. [14] developed fine-grained permission enforcement based on the context, utilizing dynamic analysis in which network permissions are given; these permissions only allow future network activity if initiated with a similar execution flow of code. They represented application context as a combination of User Identifier, Group Identifier, control flow, and data flow values and tested their implementation on the Samsung Smarthings platform. Loi et al. [18] focused on creating a systematic method to identify IoT devices' security and privacy issues by measuring the encryption protocols that were used and performing attacks against IoT devices. Like previous UI-focused approaches, these methods are coarse-grained and device-specific, and they do not differentiate between individual UI actions in the device. Furthermore, they mainly focus on network traffic in the residential environment and do not take the smartphone into account.

Unlike prior work, our approach fuses data from multiple vantage points – at the UI in the smartphone, in library/system calls from the smartphone, and in the network – to see both the smartphone and IoT devices' traffic. Our implementation supports multiple devices and is fine-grained, allowing it to differentiate between different devices and individual UI actions. This allows us to construct a causal chain of events starting from the UI and followed by protocol commands and network activity to detect anomalous traffic in real-time,

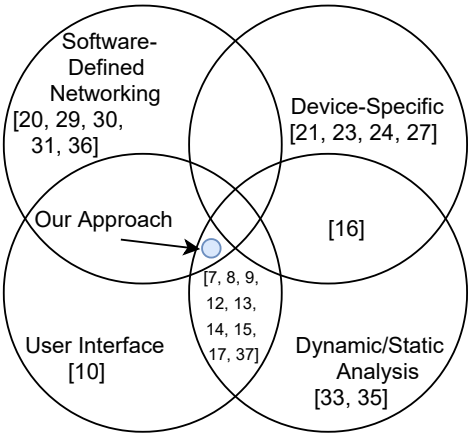

Fig. 1. Thematic comparison between our efforts and prior work

incorporating methods from past works to build a holistic view of an event. We depict our relationship with prior work in Figure 1. Our approach provides detailed context while not requiring access to the IoT device or the source code of monitored smartphone applications.

III. Approach: Fusing UI and IoT Traffic

Users often control an IoT device through a smartphone or tablet. For example, end-users can set rules, turn smart appliances on or off, set schedules, and choose various other control options depending on the IoT device type and capability. Given this common workflow, we explore methods to secure IoT devices by correlating the mobile application's UI events with the IoT device's network activity. We focus on approaches that work across mobile applications, IoT device brands, and device models. We only consider IoT home devices with defined capabilities such as locks, bulbs, and switches and do not include streaming devices such as speakers. This is to focus on devices with discrete actuation events. We seek to construct a clear causal chain of events starting from the UI event, followed by the network communication from the smartphone, and ending with the network traffic sent to the IoT device. With this sequence, we can validate network messages and ensure they are consistent with the activities of an authorized user.

Network allow-lists, which only permit known-good interactions, can significantly limit opportunities for adversaries. We explore how sensors at the smartphone and network router can help construct such allow-lists and enforce them. We further explore dynamic allow-lists, in which network traffic must 1) match known patterns associated with a legitimate request, and 2) be preceded by a user action that corresponds to the network patterns. While such tight restrictions may be infeasible in a generic computing environment (e.g., where background or non-deterministic actions are common), the network interactions with IoT devices are specifically user-driven and follow a restricted protocol. We build upon prior work that found that such protocols to be deterministic [1].

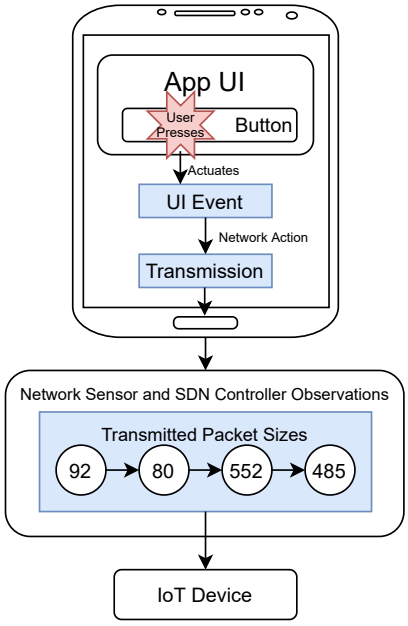

Fig. 2. IoT end-to-end activity represented as a chain of events starting from the user input, to the protocol, to a network packet sequence.

A. Threat Model

We assume the adversary has the same capabilities as the legitimate end-user in our threat model, capable of remotely accessing and controlling the IoT device using the vendor-provided smartphone application. The threat model is analogous to a situation where an adversary has acquired IoT smart applications' end-user credentials (such as via credential stuffing with reused passwords) and controls IoT devices as if they were the end-user. We assume the adversary does not have physical access to an authorized phone and thus cannot interact with its touchscreen to actuate the UI controls that our system uses to enable access.

B. Endpoint Sensors and SDN

Our approach requires sensors in both the smartphone device and in the network to observe traffic. We describe both types of sensors and the data they provide. Given Android's roughly 72% smartphone market share [15], we focus on the Android smartphone operating system.

In the endpoint, we need a sensor to monitor user interface activity. Our UI Monitor observes the information displayed to the end-users, as well as the actions that the end user takes. Since application developers must already design their UIs to be easily understood by end users, we can leverage this context to help achieve access control goals.

Android provides an `AccessibilityServices` API that allows developers to create tools that support end users with varying needs, such as screen readers for those with vision impairments. The library has been previously used by developers to help with automating UI testing and similar tools. The library is powerful because it allows applications to register callback functions for UI transitions and actions in

other applications. Further, the library provides a mechanism to allow traversal of the UI tree and to acquire the details of individual UI widget properties, such as names, class types, and displayed text. We leverage these capabilities to monitor events across devices to provide context to network requests. Importantly, this approach allows monitoring of applications on the device without requiring access to the source code of those applications.

To see the resulting network activity, we use tools from the software-defined networking (SDN) paradigm. We install Open vSwitch [24] (OVS) on the device to serve as the wireless router for the IoT device. The OVS SDN agent elevates packets in new flows to an SDN controller using the OpenFlow protocol [19]. While SDN controllers often provide SDN agents with rules to cache locally to boost performance, the traffic associated with the IoT devices we monitor tends to be both low-volume and short-lived. Accordingly, the OVS agent elevates every packet sent to or from the IoT device to the SDN controller.

The SDN controller can see network traffic associated with both the smartphone and the IoT device. Often, smartphone IoT device applications contact a third-party (associated with the IoT device manufacturer) to issue commands to the IoT device. That third-party server then relays the commands to the IoT device through a separate connection. With our network instrumentation, we can see both sides of this communication. With information from the UI Monitor as well, we can associate a UI activity with the network request sent from the smartphone and infer that subsequent packets from the third-party server to the IoT device are associated with that request. As a result, we can observe these interactions over a *training period* to observe all the UI activities and their resulting network behavior. Once a sufficient profile of activity is built, we can transition to an *enforcement period* in which the SDN controller can block packets associated with the IoT device that do not match known legitimate patterns or that lack the requisite UI activity at an associated smartphone.

C. Establishing Ground Truth

A practical deployment of the approach only requires the endpoint and network sensors described thus far. This is beneficial, since the smartphone sensors can be installed in a straightforward manner (e.g., without requiring root access to the phone or rebuilding the OS). However, to evaluate the effectiveness of the tools in this work, we require a source of ground truth data. This ground truth instrumentation would not be required in a deployed scenario.

Deployers could use the Frida [25] tool to instrument Android library and system calls. To gain this capability, Frida requires root access to the device. In our tests, we use Frida to intercept messages from the IoT device applications to determine whether messages are being sent from the application to remote servers and devices before the messages are encrypted. This allows us to confirm the inferences that we make from the network traffic and allow us to move from a "correlation" link to an actual "causal" one in our analysis. Device manufacturers or security service providers could do similar analysis to create the dynamic allow-lists proposed in this work, since this analysis can be easily performed in an Android smartphone emulator. We perform this analysis without requiring access to the source code of the IoT device applications; others can do likewise. In our actual experiments, we do not use Frida. Instead, we create policies on the assumption that highly correlated events are causal, without obtaining ground truth confirmation of that relationship. A practical deployment could do likewise. Device manufacturers or security experts could manually verify these causal relationships, if needed.

D. Privacy Implications

In creating our proof-of-concept implementation and performing our evaluation, we use devices in our own lab setting. Since our measurements are solely of software and device behavior, rather than considering human behavior, we do not involve human subjects in this study. If we did, deploying these sensors would raise privacy concerns.

While the `AccessibilityServices` API gives us the ability to monitor any application on the device, we can limit this ability only to known applications by discarding events for other applications. The UI activity can be masked so that user-entered information is not transmitted across the network. If masked, the only information that would be transmitted would be control-flow interactions in IoT smartphone applications. Such context could expose potentially private information; however, prior work has shown that such control flows can already be inferred from the network traffic [1]. We merely collect the information in advance so it can be used for access control, which necessarily must happen before the interaction completes, rather than for *ex post facto* inference analysis. With these safeguards, the privacy risks may be similar to those of a network observer, which is a concern that most security tools face.

IV. Implementation: Android and OpenFlow

For our implementation, we use the OpenFlow protocol [19] to monitor the network communication. On a laptop with six 2.6 GHz cores and 16 GBytes of memory, we host two virtual machines (VMs). The first VM is an Ubuntu 20.04 LTS VM that runs the Pox [9] controller and manages a Panda Wireless PAU09 wireless adapter that the IoT devices use for their wireless connection. The Ubuntu VM uses Open vSwitch [24] on a bridge for the wireless adapter that acts as the OpenFlow agent. The second VM is an Android emulator that runs the smartphone applications as if they were on a Pixel 2 smartphone. The laptop uses an Ethernet cable to connect to a router that provides Internet access. We provide an overview of this architecture in Figure 3. While the controller is hosted locally in this scenario, we note prior work found that the controller could be hosted remotely, such as at a nearby cloud data center, without a significant impact on latency [31].

Our approach allows us to explore multiple IoT devices and smartphone applications, since multiple IoT devices can be connected via the access point and the controller can have

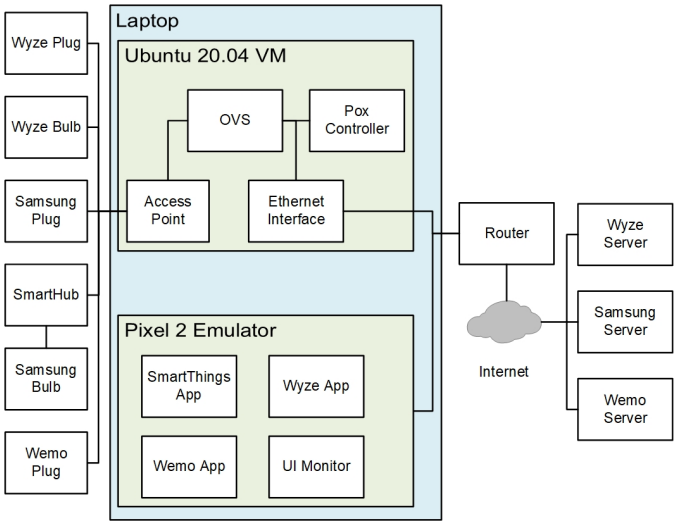

Fig. 3. System overview showing smartphone, network, IoT device, and controller components.

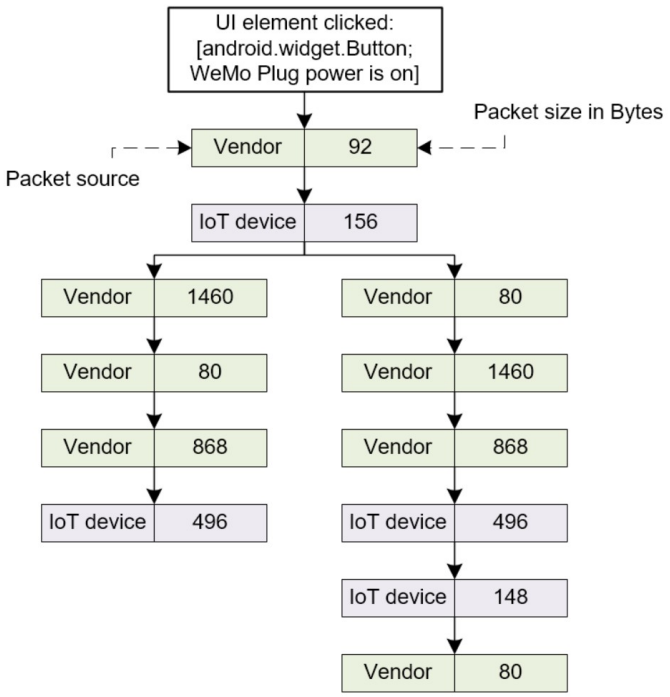

Fig. 4. An example dynamic access control policy that fuses UI activity with network packets, resulting in potential policy

a different policy for each IoT device. The module we create for the Pox controller also accepts communication from the UI Monitor running on the Android VM over UDP connections. This grants the Pox module visibility into both the smartphone application and the IoT device.

On the Android VM, we install our own application to be the UI Monitor that directly communicates with the controller. Since the UI Monitor uses the `AccessibilityServices` library, the end user must specifically enable the UI Monitor as an accessibility services provider in the Android OS settings; this step is designed to prevent malicious software from covertly monitoring user behaviors.

The UI Monitor registers callbacks with the Android OS so it can receive an `AccessibilityEvent` object whenever a UI event occurs. The `AccessibilityEvent` object contains information about the widget that is associated with the event. This information can further be used to traverse the UI hierarchy associated with that application to acquire parent `AccessibilityEvent` objects, allowing the UI Monitor to obtain information about parents and other widget ancestors. Further, we register an event handler for click events (such as a finger press). That event handler records the ID, class, text, and ancestor information associated with the actuated UI element. We leverage this data to create unique identifiers for UI widgets and to acquire descriptive contextual details (e.g., the text label associated with a button). In cases where a UI widget lacks descriptive text properties (e.g., `image_button` widgets), we acquire the text from the nearest `TextView` widget in the UI hierarchy. The UI monitor then continuously sends this UI information to the POX controller.

Figure 4 shows how we fuse the data from our sensors at the Pox controller. The policy first determines the recent UI data associated with the smartphone application. In our analysis phase, we link the UI data with the subsequent network traffic. In this example, we see two valid sequences of network packets that are permissible on the network; the differences are caused by segmentation of the network packets. In constructing the policy, we use the synchronized clock from the physical machine and then order the events based on their timestamps.

With these policies, we create allow-lists associated with each UI action. We implement the policies as state machines where the UI action is the initial state followed by a sequence of network packets as allowed states with differing packet size and server addresses based on the interactions. Some events include a delay in the network activity (e.g., activities like "turn lights on after 1 minute"), and these require special handling. In our testing, we create special time-based policies that allow network activity with delays or at specific times. We create state machines for each UI event that causes network traffic at the IoT access point. If the controller receives an elevated packet that is inconsistent with the current status of the state machine, the controller orders the OVS agent to drop the packet and all subsequent traffic in the flow. The controller also records the event as potentially malicious. This allows us to detect network activity that is unrelated to user interactions.

V. Empirical Study on Popular IoT Devices

We use experiments to explore our two research questions: *To what extent can we link end-user interactions within an IoT device's smartphone application with the resulting network flows to that IoT device? To what extent can we leverage those user interactions in filtering network traffic for IoT devices?*

These experiments examine the generalizability, effectiveness, and performance of the approach.

We conduct our experiments using consumer-grade IoT devices from three popular brands. We use common device types, such as light bulbs and power outlets, from these brands to allow cross-manufacturer comparisons. Some devices directly use WiFi for communication, while others use the Zigbee protocol to communicate with a multi-device hub that then uses WiFi to connect to the Internet. Our set of devices and manufacturers allows us to analyze both deployment models and the extent to which our techniques can generalize across manufacturers, device type, and communication protocols. We show the residential network configuration in Figure 3.

For each IoT device, we obtain the appropriate smartphone application to control the device and install it in our emulated Android device. We then manually identify a set of UI activities in that phone application that triggers network activity to the IoT device. For each such activity, we create an Appium [8] Python script that automates the associated UI activity.

While the architecture of IoT devices can vary, each tested device and brand relies upon an externally hosted server to control the IoT device. The IoT device establishes a long-lived connection with that hosting server. When we actuate UI events in the IoT devices' applications, manually or with Appium, the application contacts the manufacturers' server to send a command. That server then relays the commands to the IoT device. In the case of directly-connected WiFi devices, we observe packets sent specifically to the IoT device. For devices connected using a Zigbee hub, we observe packets en route to the Zigbee hub.

For each UI activity, we construct a policy that specifies the sequence of UI events and resulting network traffic. We use Appium to automate the actuation of these sequences to collect training data to refine our set of allow-list policies. Afterward, we again actuate these events with Appium while our Pox controller enforces these allow-lists. During the enforcement stage, we additionally test malicious traffic (which is malformed or is from a device without our application sensor). This approach allows us to gather data on generalizability, effectiveness, and performance of the approach.

The same SDN controller can manage multiple types of IoT devices simultaneously. To do so, it must have a unique state machine for each and keep track of the current state of those interactions.

A. Policy Construction and Generalizability

During our initial exploration and training phase, we use a Python script to create policies at the controller for each UI event. We represent and enforce the policies as a state machine for each device. Each state machine has a root node and events that can lead to transitions from states. For example, if the UI sensor indicates a button press, the state machine may advance to enable a new branch of network packets. Likewise, a new packet from the manufacturer's server to the IoT device may result in another transition. Each event type has its own associated data. For UI events, this includes the UI element being actuated and that element's type and identifiers. For network events, this includes the source and destination IP addresses, transport layer ports, sequence numbers, and packet size. Due to encryption, we ignore the contents of the payload and focus only on its size.

During the enforcement stage, the controller allows any network packets that can be reached from the current position in the IoT device's state machine. The controller denies any other traffic. Since the channel with the manufacturer server is multiplexed, the controller tracks and allows parallel execution of state machine branches. These actions allow background traffic, such as keep-alives, to be processed at the same time as a UI-driven event.

During our training phase, we repeatedly execute UI activity workflows until subsequent trials stop producing new traffic variants that necessitate additions to the device's state machine. In Table I, we show the UI actions and the number of different states associated with the corresponding network traffic. Each UI action can be linked to a series of network packets. While the number of states varies by device and manufacturer, we note that the technique generalizes across each device and results in a manageable size for the controller.

TABLE I
NUMBER OF STATES REQUIRED TO SUPPORT DIFFERENT DEVICE WORKFLOWS. SOME ACTIVITIES MAY HAVE STATE SEQUENCES IN COMMON.

Vendor	Device	UI Action	Number of States
Wemo	Plug	Turn On/Off	11
		Timer On/Off	16
		Vacation Mode	12
		Background	44
Samsung	Plug	Turn On/Off	6
		Background	46
Samsung	Bulb	Turn On/Off	46
		Change Brightness	102
		Background	18
Wyze	Plug	Turn On/Off	5
		Timer On/Off	4
		Vacation Mode	24
		Background	51
Wyze	Bulb	Turn On/Off	3
		Change Brightness	3
		Change Warmness	3
		Timer On/Off	9
		Background	29

B. Effectiveness Evaluation

We explore the effectiveness of our approach in terms of packet classification accuracy. We explore whether the system can allow legitimate behavior while preventing unauthorized packets from reaching the IoT device.

In our enforcement phase experiments, we use Appium [8] to randomly select and perform the UI actions from the training phase on our smartphone application. The controller receives the UI and network sensor events to determine whether to allow or deny the network packets. This process allows us to determine the approach's robustness, regardless of UI workflow order or repetition. We also insert malicious traffic by triggering network traffic on the same emulated

smartphone without our sensor reporting UI events to see if it is prevented. As described earlier, the controller examines state machines in parallel, allowing background traffic to occur at the same time as UI-driven activity. The controller could misclassify the simulated malicious traffic if it happened to be allowed by the background policy or an actuated UI-driven activity.

For each UI action associated with the five IoT devices, we collect the enforcement phase data across 1,000 trials of each action. We evaluate policy for every UI action on several IoT devices and combine them into an overall confusion matrix in Table II. The number of packets that have been correctly identified as legitimate greatly exceed those incorrectly identified as malicious with over 98% of packets being correctly classified. The system had perfect accuracy at classifying and denying malicious traffic. The approach prevents unauthorized use with minor disruption. When the controller denies packets, it can transform the denied packet into a TCP `RST` packet sent to the IoT device to cause it to disconnect and reconnect to the server. As our experiments in the next section show, this occurs quickly and restores proper operation while still filtering the undesired interaction.

TABLE II
CLASSIFICATION ACCURACY FOR TESTED SMART DEVICES.

Vendor	Device	Workflow Action	Correct Deny	Correct Allow	Incorrect Deny	Incorrect Allow
Wyze	Bulb	Turn On/Off	3,596	3,612	5	0
		Change Brightness	3,674	3,665	5	0
		Change Warmness	3,625	3,588	13	0
		Timer On/Off	7,247	7,317	68	0
Wyze	Plug	Turn On/Off	3,001	3,162	14	0
		Timer On/Off	4,054	3,977	28	0
		Vacatio n Mode	3,002	3,000	2	0
Samsung	Bulb	Turn On/Off	4,344	4,254	28	0
		Change Brightness	5,873	5,715	105	0
Samsung	Plug	Turn On/Off	2,918	3,086	5	0
Wemo	Plug	Turn On/Off	5,032	5,006	24	0
		Timer On/Off	5,667	5,958	25	0
		Vacation Mode	5,812	5,974	18	0

We note differences between the WiFi and Zigbee devices. The Zigbee bridge establishes a connection between itself and the manufacturer's server with a separate connection for each controlled device. The packets to the Samsung bulb have a significant variation in packet segmentation with small byte discrepancies. We use a cumulative sum and binning technique to account for these segmentation effects. This approach lead to high classification accuracy.

The allow-list approach ensures that unanticipated traffic is automatically dropped. This approach prevents malformed packets of unexpected sizes from being delivered to the IoT device.

C. Performance Evaluation

Our performance evaluation explores the impact of our technique on the endpoint devices and on latency in the residential network. We use the Android Profiler developer tool to analyze the resource usage of our tool on the smartphone. It reports that the CPU utilization during our experiments averaged around 1%, that energy usage is "light," and that the memory consumption of the tool is constant at 56 MBytes. With this light resource usage, our approach is unlikely to overly tax smartphones.

Next, we explore the extent to which our tool affects IoT network communication. To characterize our system's overall impact on latency, we measure the end-to-end delay introduced by our approach by comparing baseline IoT responsiveness against IoT responsiveness with our system running. We use a physical Nexus 5 phone connected to the IoT access point for this experiment. We measure the end-to-end delay using two time stamps: one taken at the initiation of the UI event, and one taken when the first packet associated with the UI event arrives at the IoT access point. The difference between these times includes the consultation with the OpenFlow controller when our approach is employed. We show the results in Figure 5. Our approach adds 20 milliseconds of delay, at most, for around 90% of traffic. Relative to the baseline, this constitutes an overhead of less than 8%. Accordingly, we believe that the delay would not be a significant concern related to usability.

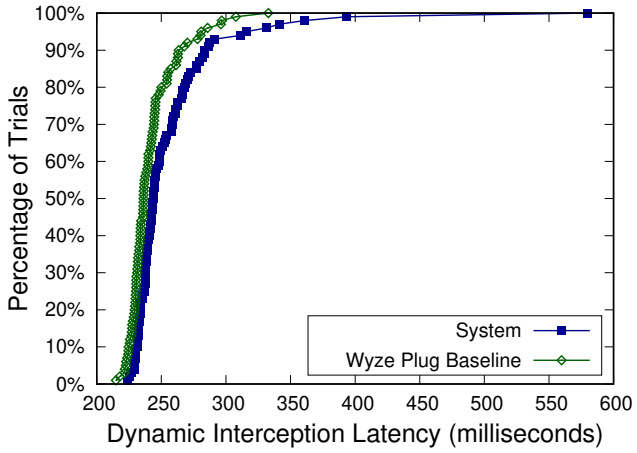

Fig. 5. Comparison between end-to-end delay of baseline and our system

We evaluate the impact of filtering malicious traffic on the device's operation. When the controller identifies a malicious packet, it transforms that packet into a TCP packet with `RST` flag and no payload. When the IoT device receives this packet, it disconnects from the server and tries to reconnect. We measure the time required for this reconnection by measuring the time from when the controller sends the `RST` packet and when it receives the `SYN` packet from the IoT device during its reconnection attempt. In Figure 6, we show the elapsed time for IoT devices from each of the three manufacturers over 1,000 trials. All three IoT devices attempt to reestablish the connection within 4,000 milliseconds. Importantly, the connection drop approach results in the unwanted action being filtered: none of the manufacturer servers retransmit the filtered action once the connection is reestablished. Accordingly, traffic can be filtered with only short disconnection periods, minimizing the impact on users' experiences on IoT devices.

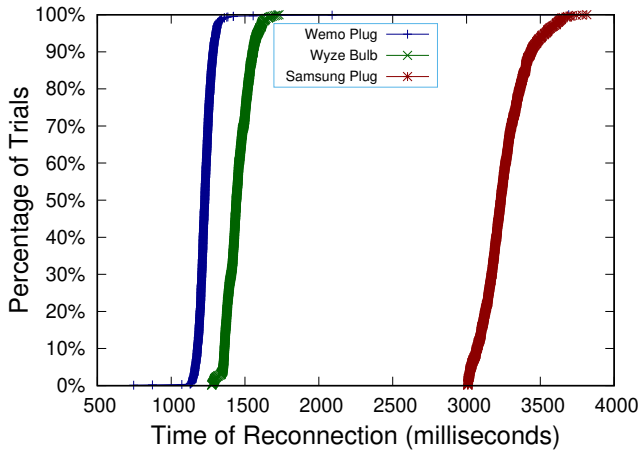

Fig. 6. Elapsed time for IoT devices reconnecting with server

In summary, these experiments show that our method is applicable to a real-world residential network. Our approach not only works well with IoT devices from different vendors and with different communication protocols; it also achieves 98% to 100% accuracy on identifying legitimate and malicious traffic, with a degree of precision that makes it difficult for an adversary to control an IoT device without being detected. Moreover, the overheads associated with the approach are low, adding less than 20 milliseconds of end-to-end delay for around 90% of traffic. Finally, even in the rare case in which traffic is incorrectly filtered, control of the device is quickly restored, allowing the user to retry the action.

VI. Concluding Remarks

We find that we can model actions of IoT devices with finite network behavior as state machines and effectively enforce dynamic allow-list policies to control access to those IoT devices. Since our system only allows known-legitimate traffic, it naturally stops anomalous traffic. This increases the challenge associated with an effective attack, since an adversary can only communicate with an IoT device after a legitimate user interaction. Further, the traffic to that IoT device must match known legitimate interactions, constraining the packet sizes and timing an adversary may use. When both legitimate and adversary traffic are sent to an IoT device, the extra traffic would not match a legitimate pattern, causing the system to filter the traffic and generate an alert.

Our experiments show that this approach works across manufacturers, device types and communication protocols. Each new device requires its own training phase; however, the resulting policy can be used across instances of the device. As a result, a manufacturer or security service provider can create policies and distribute them across SDN controllers to protect a large number of IoT devices.

Future work could extend this exploration by including additional types of IoT devices, such as media streamers or smart speakers, and different types of longer-range connectivity.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1651540.

References

[1] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. Peek-a-Boo: I see your smart home activities, even encrypted! In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 207–218, 2020.

[2] Android Developers. Application exerciser Monkey. https://developer.android.com/studio/test/monkey/, 2022.

[3] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *2010 Conference on Malicious and Unwanted Software*, pages 55–62. IEEE, 2010.

[4] Elie Bursztein. Inside the infamous Mirai IoT botnet: A retrospective analysis. https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/, Aug 2020.

[5] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. Curiousdroid: automated user interface interaction for android application analysis sandboxes. In *Conference on Financial Cryptography and Data Security*, pages 231–249. Springer, 2016.

[6] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IotFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *NDSS*, 2018.

[7] Mauro Conti, Luigi Vincenzo Mancini, Riccardo Spolaor, and Nino Vincenzo Verde. Analyzing android encrypted network traffic to identify user actions. *IEEE Transactions on Information Forensics and Security*, 11(1):114–125, 2015.

[8] Appium Developers. Introduction to appium. https://appium.io/docs/en/about-appium/intro/, 2022.

[9] POX Developers. Pox wiki. https://openflow.stanford.edu/display/ONL/POX+Wiki.html.

[10] Hao Fu, Zizhan Zheng, Aveek K Das, Parth H Pathak, Pengfei Hu, and Prasant Mohapatra. Flowintent: Detecting privacy leakage from user intention to network traffic mapping. In *IEEE Conference on Sensing, Communication, and Networking (SECON)*, pages 1–9, 2016.

[11] Andrea Gianazza, Federico Maggi, Aristide Fattori, Lorenzo Cavallaro, and Stefano Zanero. Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications. *arXiv e-prints*, pages arXiv–1402, 2014.

[12] Tianbo Gu, Zheng Fang, Allaukik Abhishek, Hao Fu, Pengfei Hu, and Prasant Mohapatra. Iotgaze: Iot security enforcement via wireless context analysis. In *IEEE Conference on Computer Communications*, pages 884–893. IEEE, 2020.

[13] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. SUPOR: Precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium (USENIX Security)*, pages 977–992, 2015.

[14] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and SJ Unviersity. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *NDSS*, 2017.

[15] Federica Laricchia. Mobile OS market share 2021. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, Feb 2022.

[16] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 3–17. IEEE, 2014.

[17] Yu Liu, Curtis R Taylor, and Craig A Shue. Authenticating endpoints and vetting connections in residential networks. In *International Conference on Computing, Networking and Communications (ICNC)*, pages 136–140. IEEE, 2019.

[18] Franco Loi, Arunan Sivanathan, Hassan Habibi Gharakheili, Adam Radford, and Vijay Sivaraman. Systematically evaluating security and privacy for consumer iot devices. In *Workshop on Internet of Things Security and Privacy*, pages 1–6, 2017.

[19] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[20] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, N Asokan, Ahmad-Reza Sadeghi, and Sasu Tarkoma. IoT Sentinel: Automated device-type identification for security enforcement in IoT. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 2177–2184. IEEE, 2017.

[21] Thien Duc Nguyen, Samuel Marchal, Markus Miettinen, Hossein Fereidooni, N Asokan, and Ahmad-Reza Sadeghi. DIoT: A federated self-learning anomaly detection system for iot. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 756–767. IEEE, 2019.

[22] Lindsey O'Donnell. 2 million iot devices vulnerable to complete takeover. https://threatpost.com/iot-devices-vulnerable-takeover/144167/, Apr 2019.

[23] Lindsey O'Donnell. Top 10 iot disasters of 2019. https://threatpost.com/top-10-iot-disasters-of-2019/151235/, Dec 2019.

[24] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 117–130, 2015.

[25] Ole André V. Ravnås. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/, Mar 2021.

[26] Security Today. The IoT rundown for 2020: Stats, risks, and solutions. https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx?Page=3, Jan 2020.

[27] Kewei Sha, Ranadheer Errabelly, Wei Wei, T Andrew Yang, and Zhiwei Wang. EdgeSec: Design of an edge layer security service to enhance IoT security. In *IEEE International Conference on Fog and Edge Computing (ICFEC)*, pages 81–88. IEEE, 2017.

[28] Graham Sharples. Iot cybersecurity threats – how cybercriminals target iot. https://blog.nettitude.com/iot-cybersecurity-threats-how-cybercriminals-target-iot-nettitude, Feb 2020.

[29] Arunan Sivanathan, Daniel Sherratt, Hassan Habibi Gharakheili, Vijay Sivaraman, and Arun Vishwanath. Low-cost flow-based security solutions for smart-home iot devices. In *IEEE Conference on Advanced Networks and Telecommunications Systems (ANTS)*, pages 1–6. IEEE, 2016.

[30] Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli, and Olivier Mehani. Network-level security and privacy control for smart-home iot devices. In *IEEE Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 163–167. IEEE, 2015.

[31] Curtis R Taylor, Tian Guo, Craig A Shue, and Mohamed E Najd. On the feasibility of cloud-based SDN controllers for residential networks. In *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–6. IEEE, 2017.

[32] Ryan Williams, Emma McMahon, Sagar Samtani, Mark Patton, and Hsinchun Chen. Identifying vulnerabilities of consumer Internet of Things (IoT) devices: A scalable approach. In *IEEE Conference on Intelligence and Security Informatics (ISI)*, pages 179–181. IEEE, 2017.

[33] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[34] Jacob Wurm, Khoa Hoang, Orlando Arias, Ahmad-Reza Sadeghi, and Yier Jin. Security analysis on consumer and industrial iot devices. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 519–524. IEEE, 2016.

[35] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *IEEE/ACM IEEE International Conference on Software Engineering*, volume 1, pages 303–313. IEEE, 2015.

[36] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things. In *ACM Workshop on Hot Topics in Networks*, pages 1–7, 2015.

[37] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. Homonit: Monitoring smart home apps from encrypted traffic. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1074–1088, 2018.