Leveraging Android OS to Secure Diverse Devices in Residential Networks

by

Shuwen Liu

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

_____

Shuwen Liu

April 2023

APPROVED:

_____

Professor Craig A. Shue, Thesis Advisor

_____

Professor Robert Walls, Thesis Reader

_____

Professor Craig A. Shue, Department Head

Abstract

As the complexity of home networks has grown in recent years, the security of multi-type devices connected in the residential environment, such as Internet-of-Things (IoT) devices, is increasingly important due to their widespread presence in homes and their physical capabilities. Many people are concerned about potential security weaknesses in these devices, since home networks lack the powerful security tools and trained personnel available in enterprise networks. However, securing various devices connected in home networks is challenging due to their different communication protocol, varied designs, multiple manufacturers, and potentially limited availability of long-term firmware updates. Home networks could introduce new physical hardware to enable security screening locally for securing in-network devices, but the capital and purchase costs may impede deployment. In this work, we propose several methods to utilize the Android smartphone operating system to secure varied devices connected in home networks.

We explore access control mechanisms that tightly constrain access to IoT devices at the residential router, with the goal of precluding access that is inconsistent with legitimate users' goals. This approach works across device types and manufacturers with a straightforward API and state machine construction. The results of our experiments show it identifies 100% of malicious traffic while correctly allowing more than 98% of legitimate network traffic. Moreover, we introduce a system to leverage existing available devices that are already inside a home network, such as Android smartphones, to create a platform for traffic inspection. This software-based solution avoids new hardware deployment and allows decryption of traffic without introducing new risk associated with new third parties. In terms of performance, we note it outperforms the state-of-the-art method in on-router traffic inspection. We also improve and evaluate an Android-based software-defined networking (SDN) system, which shows promising results on securing network traffic in Android applications.

Contents

List of Figures

List of Tables

1 Introduction

A 2016 survey shows that 77% of US households have a desktop or laptop computer and 76% percent of households have a smartphone [28]. In addition to these common personal computers and smartphones, some people deploy Internet-of-Things (IoT) devices within their home networks, such as smart light bulbs that are connected and controlled by smartphone applications. Deploying these network devices causes home network traffic to be more complicated and introduces potential security risks. In a study of fifteen people with smart home tools, eleven participants indicated they were worried about the physical risks of these devices, and five participants were concerned about the associated privacy risks [42]. To address concerns about the security of devices in home networks, we explore multiple approaches to provide practical ways to secure devices across manufacturers, types, operating systems and communication protocols.

In considering challenges for securing devices in home networks, the variety of devices connected in home networks may complicate efforts to build a comprehensive network security system. In particular, IoT devices in home networks have diverse vendors and communication protocols. Most of these devices are controlled by users' smartphone applications and rely on encrypted connections with vendors' servers. Cross-protocol and encrypted network traffic is hard to analyze in a network security system. This work aims to find a generalized and effective mechanism that works across different types and vendors of IoT devices.

Personal computers and smartphones that are common in home networks also tend to require high bandwidth, low delay and data privacy. These needs requires the network security system to inspect traffic while ensuring the network performance and privacy protection. Further, the consumer-grade routers in home networks normally are equipped with limited computing resources. With these limits, state-of-the-art methods may not work with devices that require high throughput, since the router cannot ensure enough bandwidth while inspecting network traffic.

Security concerns surrounding IoT devices in residential networks have led to significant prior work, including in the device classification [29], mobile application [2], and vulnerability analysis spaces [39]. This work has the same constraint: it attempts to secure IoT devices by focusing only on the IoT device itself. End-users typically control IoT devices remotely via smartphone applications, so we believe the user interface (UI) interactions may provide useful context for increasing IoT device security. We introduce a mechanism to fuse network traffic and UI interactions into security policies to filter traffic to IoT devices. Beyond smartphone applications for IoT devices, we explore the use of Appjudicator [26] to secure the network traffic through other popular applications on Android devices. Appjudicator works as a common Android application and does not require root privileges, unlike prior work [16]. While Appjudicator is a good foundation for securing Android devices, it still faces

several effectiveness and performance challenges.

The restricted computational capabilities of consumer-grade routers in home networks cannot fulfill the requirements for inspecting traffic and ensuring network performance. While prior work has proposed lightweight functionality on residential routers [19], the supported security tasks are severely limited. Prior work [10] has explored mechanisms to allow home networks to outsource their security and management functionality to cloud-hosted servers. However, the cloud-based solution would increase end users' privacy concerns. In contrast to prior efforts, we consider using the spare computing power of Android devices connected in home networks. After forwarding network traffic to an Android device, its computing power could be utilized to inspect traffic. With this relatively higher computational capability, these devices could support thorough vetting while ensuring high throughput in home networks.

To build a comprehensive network security system in home networks, we consider mechanisms to leverage UI interactions in access control of network traffic related to IoT devices and smartphone applications. We also explore mechanisms to deploy home network traffic inspection in an opportunistic fashion. In doing so, we ask the following research questions:

- To what extent can we link UI interactions in smartphone applications with the resulting network flows related to IoT and Android phone devices in home networks?
- To what extent can we leverage those UI interactions in filtering network traffic for IoT and Android phone devices in home networks?
- To what extent can we utilize current resources within a home network to enable real-time packet inspection?
- To what extent would such a packet inspection system influence the performance of the home network, in terms of traffic latency, resource consumption, and throughput?
- To what extent can the system that fuses UI interactions and network activities be used to predict network flows on Android devices?
- To what extent can this on-device system influence the CPU, memory, battery, and network latency on mobile devices?

In exploring these questions, we make the following contributions:

- **Implementation of a network traffic sensing system that works with varied devices in home networks:** This system is built by incorporating tools from the software-defined networking (SDN) paradigm and from Android's built-in API `VpnService` [6]. We can use the system

to sense network traffic related to all home network device despite their varied manufacturers and communication protocols.

- **Leverage of a UI sensor tool in the Android operating system (OS) to explore security mechanisms:** We use a tool built with Android `AccessibilityServices` [32] that enables us to receive and record every UI interaction that end users execute on smartphones in real time. These UI interactions provide both the user's click information and the context in which the click happens. We implement this tool in varied smartphone applications and leverage UI information to develop security mechanisms.

- **Fusion of network traffic and UI interactions for profiling behaviors of IoT devices and Android applications:** We combine the network traffic and UI interactions data in the SDN controller, and then analyze the behaviors of IoT devices and Android applications to improve profiling accuracy.

- **Creation of Router and Middlebox Support:** We explore a system to leverage existing available devices that are already inside a home network, such as smartphones and tablets, to create a platform for traffic inspection. We use open source firmware on a consumer-grade residential router. By configuring network traffic forwarding mechanisms in the router, this software-based system utilizes spare computational resources in home networks to inspect high-throughput traffic without need for new hardware deployment.

- **Evaluation of the introduced approaches' generalizability, effectiveness and performance:** After implementing the above systems to improve security of varied devices in home networks, we evaluate these approaches' generalizability by testing them in devices across types, manufacturers and communication protocols. We evaluate their effectiveness by examining the profiling accuracy rate. We evaluate the network performance of each approach in several metrics, such as throughput, end-to-end delay, and round trip time (RTT).

2 Related Work

We explore mechanisms to leverage information and resources from Android devices to secure devices in home networks. Related work in SDN shows that this paradigm works well in home networks. We choose multiple SDN tools to sense and control traffic within the local network. We also review prior work on utilizing UI interactions to improve security. We use Android `AccessibilityServices` [32] to implement a UI monitor to capture the context of all UI interactions on the Android OS. During the process of developing a security system in home networks, we analyze the performance challenges of

applying security inspection in residential routers. We discuss some state-of-the-art security techniques that can achieve high accuracy in detecting attacks on home networks but can be impractical in residential routers due to their use of computational resources.

2.1 Software-defined Networking

SDN techniques to protect residential IoT devices have been widely explored in prior work. SDN separates the control plane and data plane in networks. Taylor *et al.* [35] consider the feasibility of applying cloud-based SDN controllers in residential networks. This work shows that it is practical to utilize SDN controllers in home networks. Open vSwitch (OVS) [27] can be deployed to efficiently elevate network packets sent from virtual machines (VM). We apply these concepts in our approaches. For incorporating SDN techniques, Sivanathan *et al.* [31] propose a flow-based network defense of IoT devices with SDN techniques, and Yu *et al.* [41] propose using SDN techniques and a cloud-based service to store malicious attack signatures to protect IoT devices. Both show that SDN techniques are applicable to secure IoT devices. While those studies focus solely on network traffic in the home, our work combines network behavior with the smartphone UI interactions that caused the traffic.

While SDN has been extensively studied in enterprise networks [22] and residential networks [23], the typical implementation of SDN agents is in switches, either in hardware or in virtual hypervisors. However, SDN agents can also be deployed on endpoints, as demonstrated in previous works. For instance, Taylor *et al.* [36] provide a prototype to shift an SDN agent from a switch to an end-host, while Lei *et al.* [21] deploy SDN systems on desktops in an enterprise network for traffic engineering. Moreover, Lei *et al.* [22] show that correlated signals from endpoints can reveal compromises on those endpoint devices.

SDN has also been implemented in Android through meSDN [20] and PBS-Droid [16], which propose deploying SDN agents in the Linux kernel on Android. However, this method requires recompiling the OS, which can be a significant obstacle for many users. HanGuard [5] addresses this limitation by proposing to install SDN technology on both Android endpoints and home routers to enforce fine-grained access control. In contrast, our work focuses on improving deployability and enabling greater context from endpoints to enhance access control decisions.

2.2 UI Interactions and Accessibility

Prior works explore the functionality of UI interactions in identifying legitimate behaviors on multiple platforms. SUPOR uses static analysis to identify UI interactions that include sensitive user inputs as potential privacy or security risks [17]. Gianazza *et al.* [12] use recorded UI interactions from malicious applications to determine if similar UI interactions yield malicious behavior in other Android

applications. Beyond leveraging UI interactions to directly detect malicious behavior, Harbinger [4] identifies human-initiated network traffic in the Window OS with 99.1% accuracy while adding less than 6 milliseconds of delay. Zorig *et al.* [3] introduces a tool that supports leveraging UI interactions to secure IoT devices and Android phones. Unlike prior work, our approach fuses data from multiple network devices to see both the Android phone and IoT devices' traffic. Our implementation supports varied devices and is sufficiently fine-grained to differentiate between various devices and individual UI interactions. This allows us to construct a causal chain of events starting from the UI and following protocol commands and network activity to detect anomalous traffic in real-time.

Mobile device end-users interact with programs via the user interface, which provides context for different operations. However, existing techniques for real-time traffic classification have limitations. For instance, AppIntent [40] utilizes static analysis to trace UI activities to data transmission events, while the GUILeak [38] tool traces UI actions and manually compares user input with contractual and privacy policies described by the vendor to detect privacy violations. These techniques are not suitable for real-time traffic evaluation due to the manual steps involved. Moreover, SmartDroid [43] proposes to enforce a sequence of system calls associated with UI elements, but it requires rooting the device, making it difficult to implement on average users' mobile phones.

Previous research has leveraged the UI with traces and logs to identify application defects and to localize issues [34]. To aid software testing, tools have been developed to automate UI interactions. For example, the Android `Monkey` [33] tool allows developers to send sequential events to the device via its `adb` debugger interface, which we use in our experiments to examine hierarchical UI elements. Additionally, Android's accessibility API [32] enables accessibility service developers to cross the traditional boundary of sandboxed applications. These services can send commands and receive UI events invoked by a human user from other applications.

2.3 Security Techniques in Residential Networks

Prior work [1, 14] indicates that modern home networks face many security risks. Attackers can gain sensitive information or directly control the devices and launch attacks on other devices, such as eavesdropping, replay attack, network scanning, and data theft. There are effective ways to detect these attacks, but they require sufficient computational resources. Hafeez *et al.* [14] find that machine learning methods can detect a series of attacks with accuracy as high as 99%. Jan *et al.* [18] propose a method based on a deep learning algorithm to detect a compromised device that has joined a botnet. In this work, we aim to create a system utilizing spare computational resources in Android devices to enable such traffic inspection in home networks.

Given the limitations of residential routers, some prior work explored mechanisms to shift the com-

putational tasks associated with network screening to remote servers. Feamster [10] proposes using SDN techniques to allow home networks to outsource their security and management functionality to cloud-hosted servers. Likewise, TLSDeputy [37] uses remote servers to validate the TLS certificates and protocol settings associated with home network connections to ensure the authenticity of communicating endpoints. However, both techniques allow the operators of cloud infrastructure to have insight into the activities of a home network, introducing new privacy risks. Gedeon *et al.* [11] propose to use a local device, such as a home network gateway, to run a broker to coordinate tasks. The gateway seeks available cloudlet nodes to help with its tasks. This method outsources computation to third-party platforms, which raises privacy concerns. However, its use of a broker on a residential router, and the finding that it does not introduce significant overhead, is useful for our own approach. We explore a platform that enables network traffic inspection within a home network. Unlike existing cloud-based solutions, we focus on streaming tasks that require continual computation rather than discrete computational jobs.

In this work, we implement Appjudicator [26] which is an Android application that can utilize UI interactions to improve network profiling accuracy. It plays an important role in our research. As shown in Figure 1, when a user clicks a button on a phone screen, the Appjudicator uses Android accessibility service [32] to capture the UI interaction. If the UI interaction results in any network flow, Appjudicator captures it with its Android VPN [6] client. Appjudicator also acts an an SDN agent and submits these two pieces of information to an SDN controller. The controller makes better decisions with the combination of UI interactions and network flows. While Appjudicator is a good foundation for Android security research, it still has several issues regarding stability and performance. In our research, we address these issues and evaluate its effectiveness and performance in multiple experiments.

3 Securing Multi-type IoT Devices

IoT devices have entered the mass-market adoption phase in the residential environment, particularly consumer electronics such as smart power outlets, light bulbs, and media-streaming devices. These multi-type devices are made from varied manufacturers with different operating systems and communication protocols. They are often embedded into a home for years 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 varies 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 users.

In investigating the vulnerabilities in IoT devices, we found a common user behavior that could be

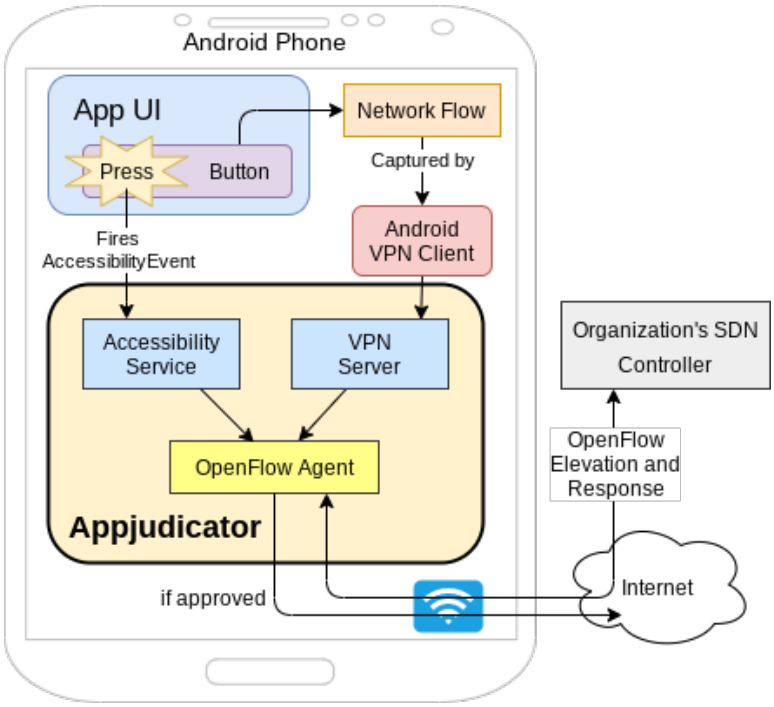


Figure 1: System Overview

used to develop a general mechanism to enhance 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 UI interactions and network packets 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.

In this Section, we develop mechanisms to leverage UI interactions in access control of network traffic related to IoT devices and smartphone applications. We incorporate tools from the SDN paradigm to conduct the access control. We run several experiments to evaluate the generalizability, effectiveness, and performance of the approach. By doing so, we want to answer the following research questions:

- To what extent can we link UI interactions in smartphone applications with the resulting network flows related to IoT and Android phone devices in home networks?
- To what extent can we leverage those UI interactions in filtering network traffic for IoT and Android phone devices in home networks?

3.1 Fusing UI and IoT Devices Traffic

In this Section, we explore methods to secure IoT devices by correlating the smartphone application's UI actions with the IoT device's network activity. We focus on approaches that work across smartphone applications, IoT device manufacturers, and device types.

3.1.1 Threat Model

We build a threat model based on our understanding of user cases of IoT devices. 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. 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, continuing with 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 that they are consistent with the activities of an authorized user.

In our threat model, we assume the adversary has the same capabilities as the legitimate end-user, capable of remotely accessing and controlling the IoT device using the vendor-provided smartphone application. This 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 simulate a model where the malicious users send the commands to vendors' servers, and then the remote servers connect with IoT devices for executing the commands. 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. On the side of legitimate users, we consider they could use their smartphone outside of home networks to control the IoT devices. Our system would work well in this scenario, since it is designed to be installed on users' smartphone.

3.1.2 Endpoint Sensors and SDN

To fuse UI actions and IoT device traffic,, our approach requires sensors in both the smartphone and in the network to observe the traffic. In the endpoint, we need a sensor to monitor user interface activity. We use the `AccessibilityServices` API from Android to build a UI monitor that collects the information displayed to the end-users, as well as the actions that the end user takes. Since application developers must already design their UI items to be easily understood by end users, we can leverage this context to help achieve access control goals.

To see the resulting network traffic, we use tools from the software-defined networking (SDN) paradigm. We install Open vSwitch [27] (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 [25]. While SDN controllers often provide SDN agents with locally cached rules 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.

We fuse UI action and network traffic information in the SDN controller. The controller can see network traffic associated with both the smartphone and the IoT device. When we analyze traffic in the controller, we note that smartphone applications of IoT devices always 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 configuration, we can see both sides of this communication. With information from the UI Monitor as well, we can associate a UI action 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 see 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.

3.1.3 Implementation of Android and Openflow

For our implementation, we use the OpenFlow protocol [25] 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 [8] controller and manages a Panda Wireless PAU09 wireless adapter that the IoT devices use for their wireless connection. The Ubuntu VM uses Open vSwitch [27] 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 2. While the controller is hosted locally in this scenario, we note that prior work finds that the controller could be hosted remotely, such as at a nearby cloud data center, without a significant impact on latency [35].

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 a different policy for each IoT device. The module we create for the Pox controller also accepts communication from the

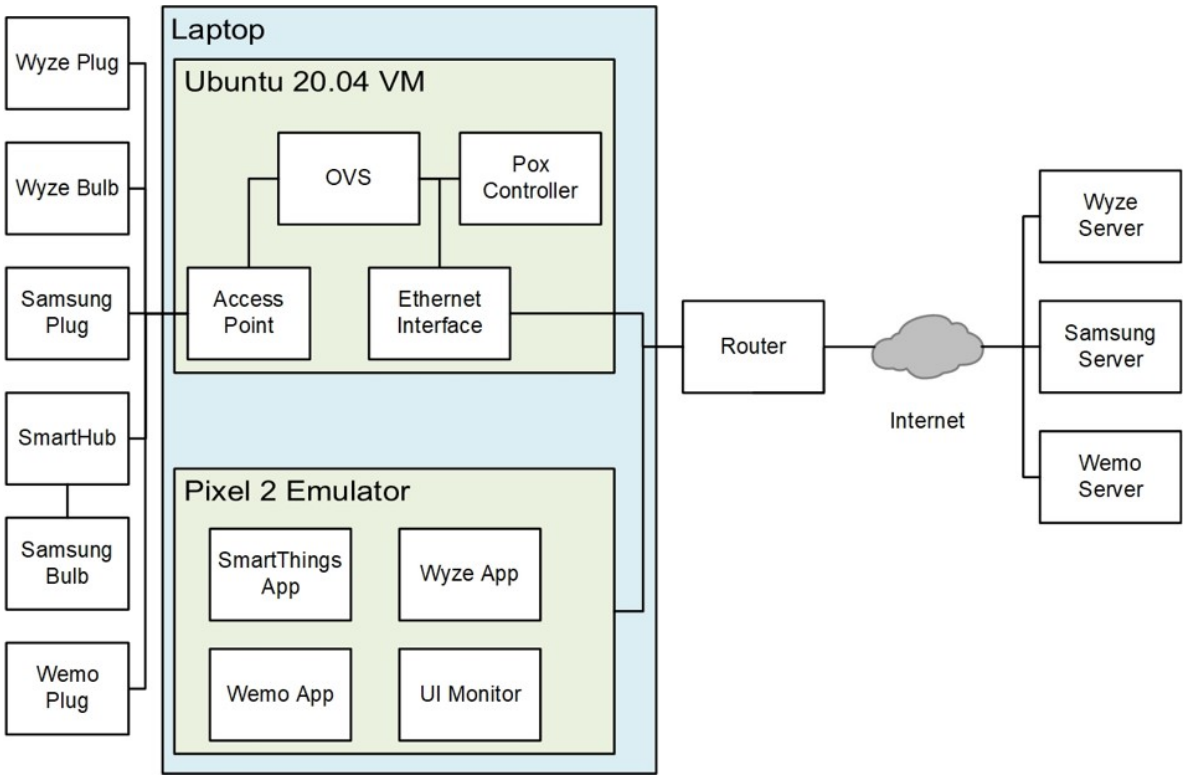


Figure 2: System overview showing smartphone, network, IoT device, and controller components.

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.

Figure 3 shows our method of fusing the data from our sensors at the Pox controller. This policy controls the connection between the vendor's server and the IoT device. It first determines the recent UI data associated with the smartphone application, so that every policy would be related to a specific UI action. 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. To be more specifically, in this policy, a network packet of size 92 bytes would be allowed only if there is a click of turning on the plug at the very beginning. 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

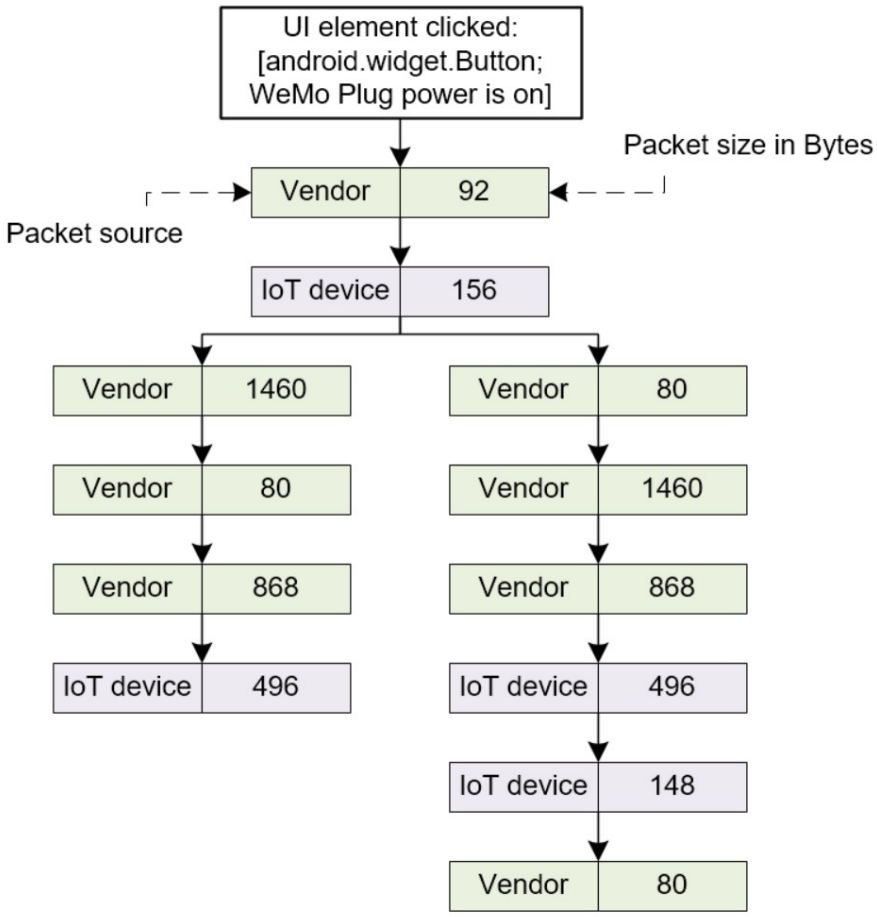


Figure 3: An example of dynamic access control policy that fuses UI activity with network packets, resulting in potential policy

as state machines where the UI action is the initial state followed by a sequence of network packets as allowed states with differing packet sizes 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.

3.2 Evaluating Proposed Approach

We use experiments to explore our two research questions:

- To what extent can we link UI interactions in smartphone applications with the resulting network flows related to IoT and Android phone devices in home networks?
- To what extent can we leverage those UI interactions in filtering network traffic for IoT and Android phone devices in home networks?

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 2.

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 use several Appium [7] Python scripts that automate 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 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.

3.2.1 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 1, 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 1: 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

3.2.2 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 [7] 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 2. The number of packets that have been correctly identified as legitimate greatly exceed those incorrectly identified as malicious, with over 98% of packets 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 2: Classification accuracy for tested smart devices.

Vendor	Device	Workflow Action	Correct		Incorrect	
			Deny	Allow	Deny	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
		Vacation 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 con-

nection 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 leads to high classification accuracy. The allow-list approach ensures that unanticipated traffic is automatically dropped and prevents malformed packets of unexpected sizes from being delivered to the IoT device.

3.2.3 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 4. 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.

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 5, 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.

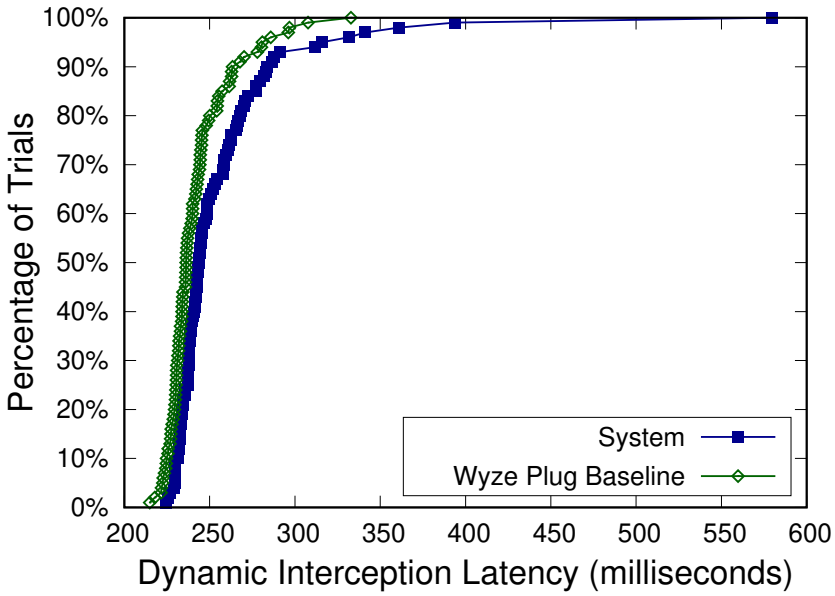


Figure 4: Comparison between end-to-end delay of baseline and our system

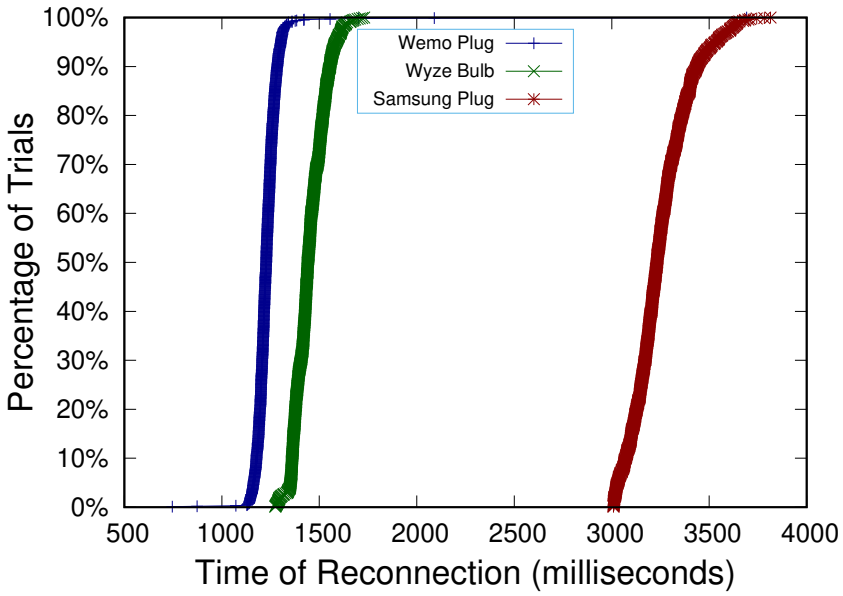


Figure 5: Elapsed time for IoT devices reconnecting with server

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 in identifying legitimate and malicious traffic. This degree of precision 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.

3.3 Summary of Section

For the first research question, the answer is the UI interactions in smartphone applications can be linked with the resulting network flows for all of tested IoT devices. 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.

For the second research question, we note the UI interactions can be used to effectively filter network traffic for IoT devices. In Chapter 3, we find that we can model IoT device actions 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 malicious 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.

4 Inspecting Traffic with Opportunistically Outsourced Middleboxes

As mentioned earlier, residential networks have grown increasingly complicated. Residential users have expressed concern about the risk from connected devices in their networks. In a study of fifteen people with smart home tools, eleven participants indicated they were worried about physical risks of these devices and five participants were concerned about the associated privacy risks [42]. While end-point solutions, such as anti-virus and software firewalls, are effective for some devices, they are not available for others. Therefore, we explore the inspection of network traffic related to home network devices, which could be a general way working with diverse devices. In this Section, we introduce the possible solution of opportunistically outsourced middleboxes to enable users to inspect their home network

traffic with consumer-grade routers.

In-network security controls, such as screening on routers or middleboxes, can help protect home network devices from network-borne threats, but current consumer-grade routers do not effectively manage network risk [24]. Due to limited computational capabilities. While prior work has proposed lightweight functionality on residential routers [19], there are inherent limits on the tasks these routers can perform. We therefore propose a method that offloads traffic inspection to other devices in the local network, under the hypothesis that redirecting network traffic to a locally available device with greater computational resources, while limiting the router's work to traffic forwarding, may yield better performance than attempting to perform the inspection on the router itself. We compare this method, which involves on-phone inspection via NAT redirection, to on-router inspection in the following experiments.

In this Section, we consider methods to deploy home network traffic inspection in an opportunistic fashion. We explore mechanisms to leverage existing devices in a home network when they are available to network communication. By doing so, we aim to answer the following research questions:

- To what extent can we utilize current resources within a home network to enable real-time packet inspection?
- To what extent would such a packet inspection system influence the performance of the home network, in terms of traffic latency, resource consumption, and throughput?

4.1 Redirecting Network Traffic through NAT Rules

In this subsection, we describe our proposed mechanisms in detail. Our method forwards network traffic from a router to a middlebox to leverage its spare computational resources. The experimental network uses devices such as smartphones, tablets, laptops and desktops to perform traffic analysis. These devices can operate as security proxies when they are available, enabling detailed analysis. We use open source firmware on a consumer-grade residential router. We use simple IP-address based screening as a conservative example of the computational requirements of security tools. We build tools to screen traffic locally on a consumer-grade router to establish a comparison model. We then implement a technique to forward network traffic through a smartphone middlebox using network address translation (NAT) rules on the router.

4.1.1 Threat Model and Scope

When exploring technologies related to security, a model of the assumed threats can determine the applicability of the work and its scope. We first construct a platform that is designed to enable the

inspection of traffic that crosses the boundary of a home network. Our platform's goal is to provide computational resources for inspection tools while achieving reasonable performance. We do not seek to develop a new detection algorithm or technique. Instead, we use a particularly lightweight filter, based on packet addresses, to demonstrate the minimal overhead costs associated with each.

A trusted computing base (TCB) is the set of devices that must operate correctly to achieve the desired security goals. Our TCB includes the residential router and any smartphones hosting middleboxes that proxy traffic. We do not need to trust the communicating endpoint device, the remote machine it is communicating with, or other infrastructure associated with the Internet. Unlike approaches that outsource communication, our TCB does not include third-party cloud servers or the personnel associated with cloud data centers. In this model, the main goal of adversaries is to build connection with a device inside home network. Both the remote server and the endpoints in home network could be potentially compromised.

4.1.2 Two Solutions for Inspecting Home Network Traffic

We compare two approaches: on-router inspection via `NFQUEUE` and on-phone inspection via NAT redirection. We start by introducing the process of on-router inspection. Then, we describe the functionalities of each component of the phone-based inspection platform and how they work together.

We create an approach that is designed to provide efficient on-router traffic inspection. We implement a basic C++ program that we compile to run natively on the router to inspect IP addresses. The program uses the `iptables` packet inspection tool and the `netfilter_queue` library (often referred to as `NFQUEUE`) to inspect traffic. Essentially, the `iptables` tool operates on each packet processed by the Linux stack on the router. This action occurs when a packet crosses from the LAN interfaces to the WAN interface associated with the Internet. The `iptables` program sets an `NFQUEUE` judgment for all packets, causing them to enter a kernel queue data structure. The C++ program extracts the packets from that queue, inspects the destination network address, and returns the packets to the kernel queue for transmission. This program represents the minimum inspection required for a general-purpose user-space inspection program on the router.

There are two components that support our traffic inspection on a separate smartphone. The first is a set of NAT rules on the router that will appropriately forward the traffic. For this, we use the `iptables` program, which can manage IP packet rules in the Linux kernel. We use the `iptables` NAT table to implement translation rules that transform the original destination IP address of the packets from the server to the IP address of the smartphone. This causes the traffic sent from the client to be redirected to the smartphone. In the example shown in Figure 6, we first apply an SNAT rule as `iptables -t nat -A POSTROUTING -p tcp -s 192.168.1.2 -d 192.168.1.193 --dport 6666 -j SNAT --to-source 192.168.1.1` and a DNAT rule as `iptables -t nat -A PREROUTING -p tcp`

`-s 192.168.1.2 -d 172.16.1.2 --dport 6666 -j DNAT --to-destination 192.168.1.3:6666` to forward traffic to the smartphone. The smartphone can then work as a proxy that receives packets and sends them back to the router after inspection. When these packets return to the router, the router transforms their destination IP address to the original server destination IP address based on another DNAT rule, such as `iptables -t nat -A PREROUTING -p tcp -s 192.168.1.3 -d 192.168.1.1 --sport 7777 -j DNAT --to-destination 172.16.57.216:6666`. Since these NAT rules function bidirectionally, the packets sent from the server will traverse the reverse path through the smartphone. Rather than processing traffic as an arbitrary user space program in the router's Linux stack, our method forwards them using kernel data structures. This feature avoids potentially costly transitions to user space on the router.

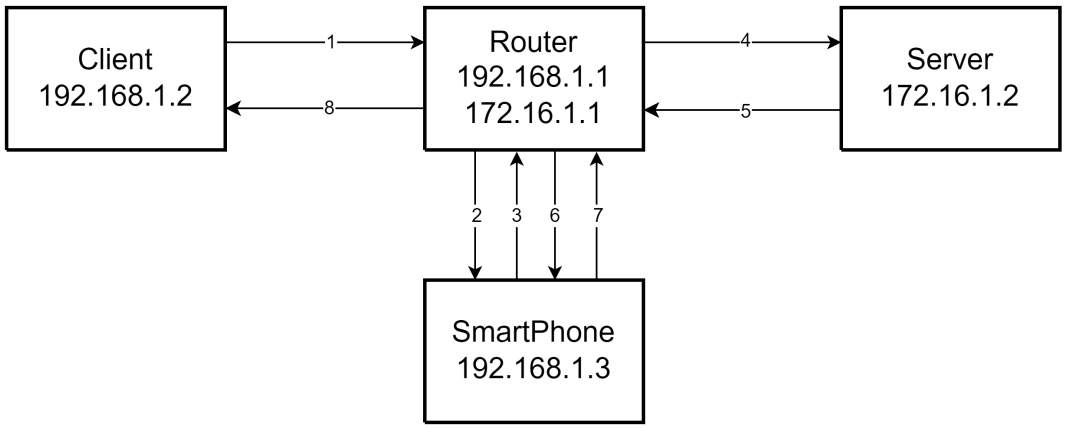
The second component in our approach is the proxy software and service that runs on the smartphone. We implement a Java program that uses TCP to accept traffic for inspection on a pre-defined port. Figure 6 shows how the phone accepts traffic from the router using a new TCP connection. Since the smartphone is on the network path between the communicating endpoints, it receives the raw payload of every network packet. While we only apply IP list filtering in our tests, more advanced inspection can be deployed in our method, such as TLS inspection. The following performance evaluations show even this common lightweight operation saturates the router with on-device inspection, whereas our NAT approach provides headroom. This approach can support more computationally demanding use cases without requiring new physical hardware deployments.

4.1.3 Implementation

We implement our method in a lab environment. We run the OpenWrt 21.02.2 operating system (OS) on a consumer-grade TP-LINK AC1750 Wireless Dual Band Gigabit Router. We simulate a home network user with a client on a laptop with four cores and 16 GBytes of memory, running the Windows OS. We simulate a server outside of the home network on a laptop with four cores and 16 GBytes of memory, running the Ubuntu 20.04 OS. We use a smartphone with eight 2.0 GHz cores and 4 GBytes of memory, running the Android 11 OS as the proxy device.

For the network configuration, as shown in Figure 7, we create two VLANs: one is on interface `eth0` and the other is on interface `eth1`. We assign the LAN ports and wireless radio to the `eth0` VLAN and assign the WAN port to the `eth1` VLAN. The client connects to a LAN port via a category 6 Ethernet cable that supports full-duplex gigabit throughput. The server also connects to the WAN port using a category 6 cable. For the radio, we build an access point on 5.785 GHz using a Qualcomm Atheros QCA 9880 802.11ac adapter. We connect the smartphone to this access point at a distance of 3 feet with an unobstructed, line-of-sight path.

After configuring the home network, we add three NAT rules to `iptables` in the router, as described



Packet #	Source IP	Source Port	Destination IP	Destination Port
1	192.168.1.2	47289	172.16.1.2	6666
2	192.168.1.1	6001	192.168.1.3	6666
3	192.168.1.3	7777	192.168.1.1	6001
4	172.16.1.1	6002	172.16.1.2	6666
5	172.16.1.2	6666	172.16.1.1	6002
6	192.168.1.1	6001	192.168.1.3	7777
7	192.168.1.3	6666	192.168.1.1	6001
8	172.16.1.2	6666	192.168.1.2	47289

Figure 6: An example of packet forwarding via NAT rules. As the client sends the original packet to the server, the router modifies the packet and forwards it to the smartphone. After the smartphone performs packet inspection, it sends the packet back to the router. Then the router forwards it to the server. Since all of the NAT rules work bidirectionally, the packets sent from the server will follow the reverse path.

in the previous subsection. These rules include SNAT and DNAT rules and have the capability of redirecting traffic between the client and the server to traverse the smartphone. On the smartphone side, we use Android Studio to build a Java application that performs packet inspection based on a malicious IP block list and hosts a proxy service.

4.2 Comparing Network Performance in Three Scenarios

An on-router inspection module is straightforward, since it uses a device that is already physically on the network path between communicating endpoints. To justify the added complexity of opportunistic middleboxes, we explore the performance implications of using such commodity devices. We first establish a baseline for the performance of the home network. We use a typical network setting, without the use of inspection functionality, to establish the baseline. We then explore on-router inspection using a simple block-listing application on a router. Finally, we examine an inspection method in which NAT rules are used to reroute traffic to a middlebox, using both a smartphone

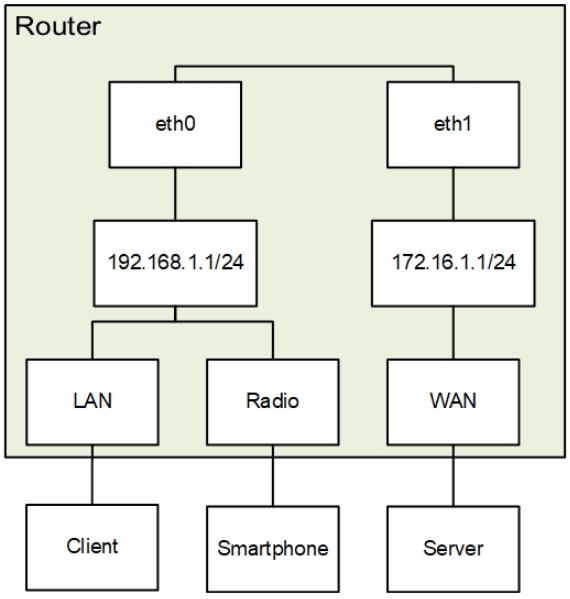


Figure 7: The network configuration for our experiments

emulator and a physical commodity smartphone for analysis.

In examining these scenarios, we evaluate the performance of each using four metrics: flow throughput, end-to-end round trip time (RTT), the CPU usage at the router, and the CPU usage of the smartphone when it is in use. We note that there is performance improvement in using on-phone inspection after comparing the three scenarios with these four metrics.

4.2.1 The Baseline: LAN to WAN traffic

Our baseline scenario connects a client to a server though a residential router. Often, the WAN port is used on the router to connect to upstream networks, such as the Internet, and the servers available through those networks. Therefore, we connect an Ubuntu server to the WAN port of the router using a full-duplex category 6 Ethernet cable. The server uses a gigabit Ethernet card. We statically configure the IP addresses of the server and the router's WAN port within a subnet that is only used by those two devices.

The connectivity options for clients may vary in different homes. Some devices may be connected via Ethernet connections to the LAN ports of the router. In other cases, devices may connect using WiFi radio links. Accordingly, we explore both of these connection scenarios.

We begin by exploring the case in which the client is connected to a LAN port on the router via a category 6 Ethernet cable. We use the router's built-in DHCP server, which assigns an address to the client in a subnet that the router and client share, yet is disjoint from the subnet used by the server. We use the router's default NAT capabilities to translate across the subnets, which is a common

deployment scenario in homes. Using the `iperf3` benchmarking tool [9], we test a TCP connection between the client and the server. We configure `iperf3` to attempt to maximize throughput in the channel and observe it for 1,100 seconds. We conducted 3 trials and measured the throughput for 1,000 seconds after an initial delay of 100 seconds to accommodate TCP's slow-start behavior. As we see in the right-most two lines in Figure 8, the median download throughput is around 440 Mbps and the median upload throughput is around 254 Mbps, with tight distributions (the standard deviation is 4.90 Mbps for download throughput and 3.27 Mbps for upload throughput).

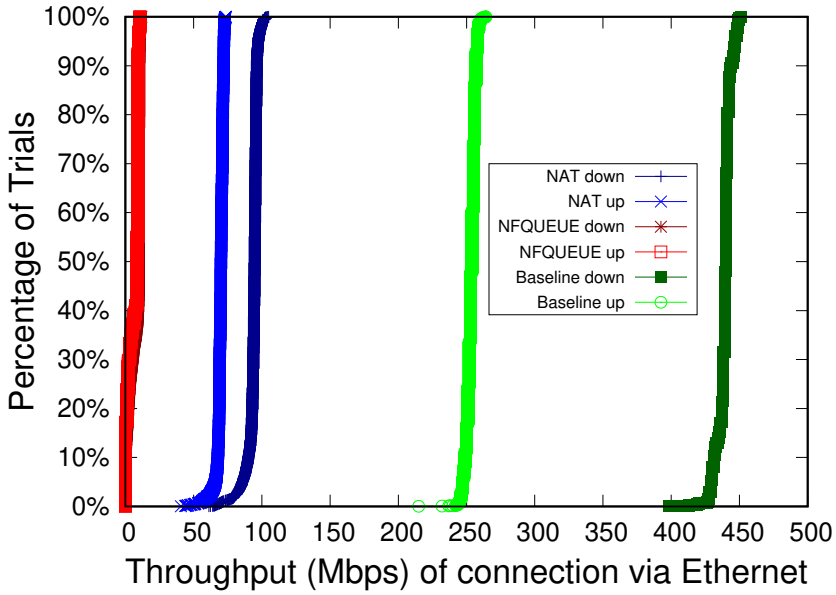


Figure 8: Results from throughput tests when the client connects to the router via a category 6 Ethernet cable. The green lines show upload and download throughput under a baseline setting. The red lines show the throughput after applying on-router inspection via `NFQUEUE` library. The blue lines show the throughput after applying on-phone inspection using NAT redirection rules.

Next, we determine the impact of connecting the client to the router using WiFi radios. Both the client and the router support 802.11ac communication, which has a theoretical maximum throughput of 1300 Mbps, though practical throughput can be lower due to interference and obstructions. We place the router and the client roughly 3 feet apart with a line-of-sight path. We then repeat our throughput analysis using the `iperf3` benchmark tool with the same settings as our Ethernet experiments. We use the same 3 trials and timing windows as in the earlier experiment. In Figure 9, the right-most two lines show the baseline throughput results via WiFi. The median download throughput is around 196 Mbps and the median upload throughput is around 229 Mbps, with a 10.38 Mbps standard deviation for download throughput and 14.47 Mbps for upload throughput.

Since the communication throughput via Ethernet appears to be less than the medium's theoretical maximum, we explore whether the router could be causing a bottleneck. In particular, we examine

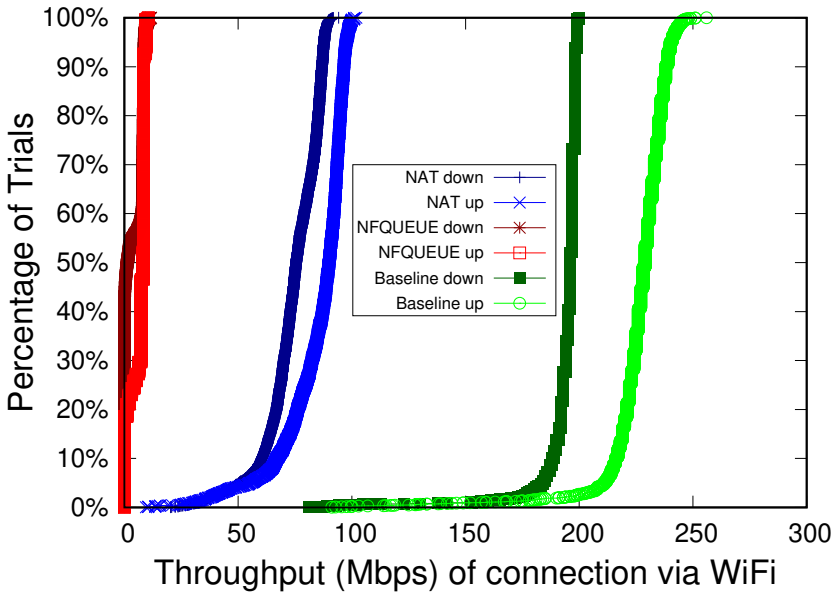


Figure 9: Results from throughput tests when the client connects to the router wirelessly. The rightmost green lines show upload and download throughput under a baseline setting. The leftmost red lines show the throughput after applying on-router inspection via `NFQUEUE` library. The middle blue lines show the throughput after applying on-phone inspection using NAT redirection rules.

the CPU of the router. While we test the maximum throughput, we use the `top` tool to record the CPU usage of the router for 1000 seconds. As shown in Table 3, the CPU usage of the router is at its limit more than 90% of the time when testing maximum throughput. These results suggest that the CPU of our router acts as a performance bottleneck when throughput is high.

Table 3: CPU usage of the router while testing the maximum throughput in six scenarios.

Percentile of Trials	10th	50th	90th
CPU Usage in Baseline Upload	100%	100%	100%
CPU Usage in Baseline Download	100%	100%	100%
CPU Usage in NAT Upload	98%	100%	100%
CPU Usage in NAT Download	97%	100%	100%
CPU Usage in NFQUEUE Upload	100%	100%	100%
CPU Usage in NFQUEUE Download	100%	100%	100%

To determine the added CPU usage from different traffic inspection methods, we need to measure the router's CPU usage in a moderate throughput scenario, rather than when throughput is maximized. We thus evaluate the scenario in which the TCP connection throughput is reduced to 10 Mbps of randomized payload to the server. We also record the CPU usage of the router for 1,000 seconds. The green line in Figure 10 shows that the median CPU usage of the router is 9.00% with standard deviation of 1.58%.

While throughput is an important metric, the end-to-end round trip time (RTT) is also important

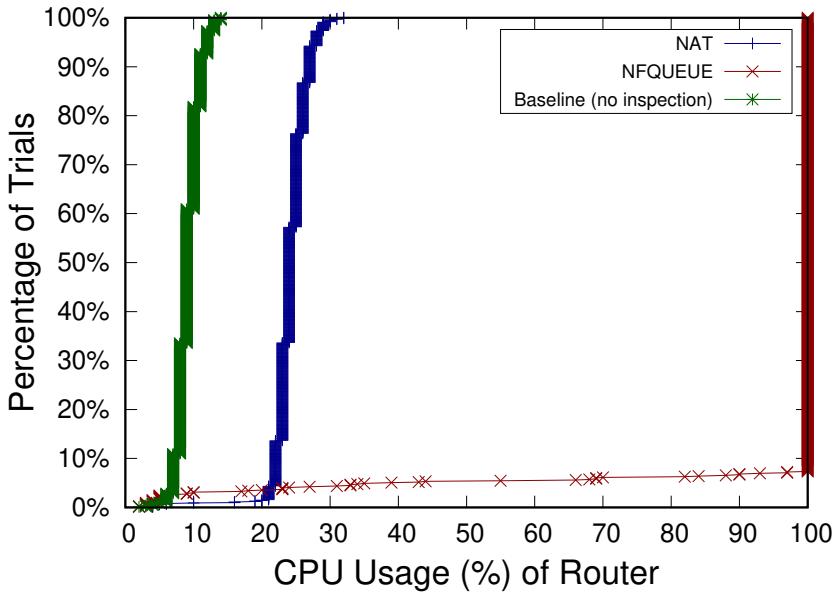


Figure 10: CPU usage of the router when applying on-phone inspection using NAT redirection rules, applying on-router inspection via `NFQUEUE` library, and a baseline without inspection when throughput is limited to 10 Mbps.

for understanding the delay introduced by the network paths and the router. To test this, we construct an echo program on the server and a recording device on the client to measure the time difference between the client sending a specific payload and receiving a reply. Across 1,000 trials, we see that the left-most line in Figure 11 has a median RTT of 1.12 ms with a standard deviation of 0.12 ms. When repeating this analysis via WiFi, in Figure 12, we see the median RTT of the left-most line is 2.72 ms with a 6.14 ms standard deviation.

4.2.2 On-Router Inspection via `NFQUEUE`

To explore whether the router itself can feasibly inspect traffic, we implement a basic C++ program, that is compiled to run natively on the router, to inspect IP addresses.

We explore the throughput, RTT, and router CPU metrics of the on-device inspection program using the same tools and settings used in the previous subsection of baseline. In the two left-most lines of Figures 8 and 9, we see the upload and download throughput after applying this inspection approach. We conducted 3 trials and measured the throughput for 1,000 seconds after an initial delay of 100 seconds to accommodate TCP's slow-start behavior. As seen in the right-most two lines in Figure 8, the median download throughput is 9.62 Mbps, and the median upload throughput is 8.40 Mbps (with a standard deviation of 3.91 Mbps for download and 3.93 Mbps for upload). Given this substantially decreased throughput from the baseline, we hypothesize that the change introduces a bottleneck on the router.

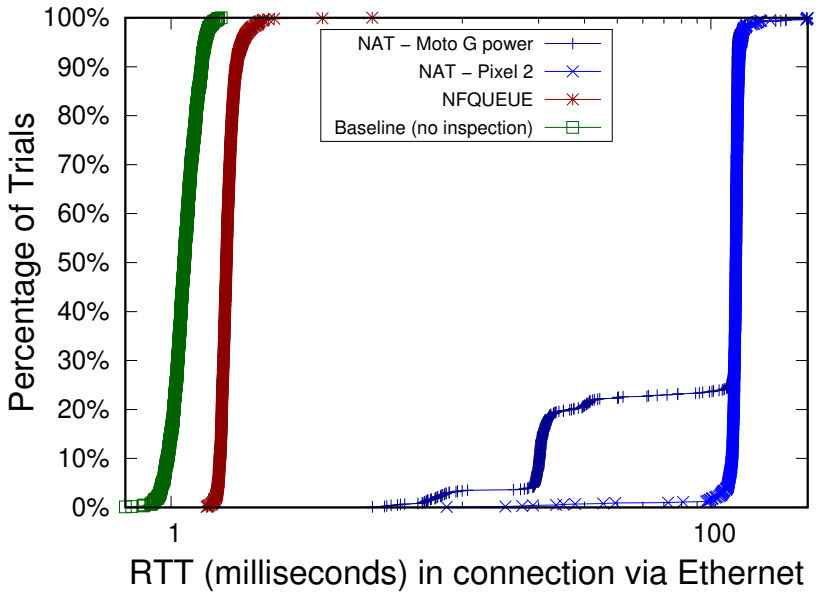


Figure 11: RTT with a log scale in milliseconds between the client and the server when the client connects to the router via Ethernet. The leftmost green line shows baseline result. The middle red line shows the result after applying on-router inspection via the `NFQUEUE` library. The two rightmost blue lines show the results with two separate phones after applying on-phone inspection using NAT redirection rules.

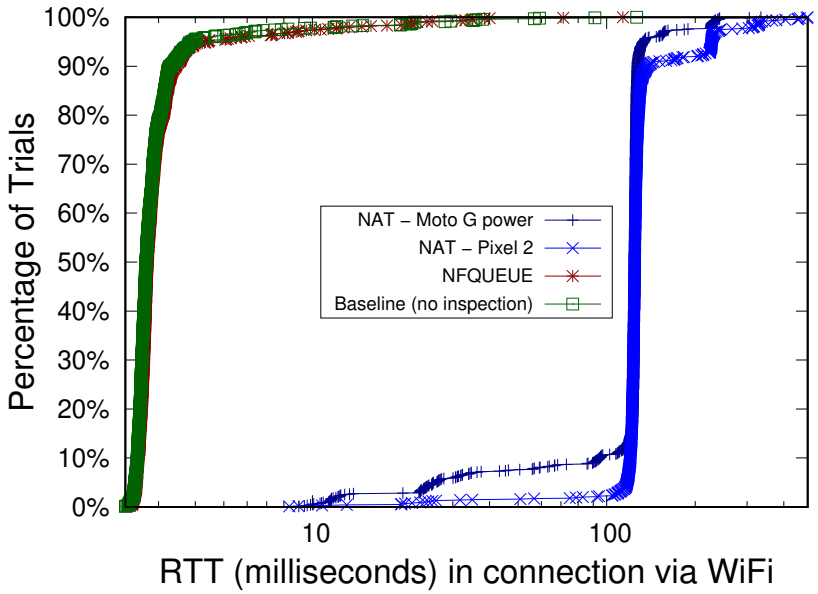


Figure 12: The RTT with a log scale in milliseconds between the client and the server when the client connects to the router via WiFi. The green line shows the baseline result. The red line shows the result after applying on-router inspection via `NFQUEUE` library. The two blue lines show the results with two separate phones after applying on-phone inspection using NAT redirection rules.

When we examine the CPU usage of the router, we confirm that this resource is exhausted. In Figure 10, we see that the baseline CPU usage is around 9% when throughput is limited to 10 Mbps,

but is 100% when the router performs packet inspection. The process elevates all traffic to the router's Linux user space environment, which requires significant computational resources on the router. Such routers tend to be manufactured with lower-end CPUs for economic reasons [15] and there appears to be little headroom for this additional operation. However, when the router is not overwhelmed, as in the simple echo server RTT tests, we see that the on-device router introduces minimal RTT increases over the baseline. These results are shown by the red line in Figure 11, which is close to the baseline results.

4.2.3 On-Phone Inspection via NAT Redirection

Due to the CPU limitations of residential routers, we explore the potential of re-routing packets via a smartphone to inspect traffic. As described in the previous subsection, we add three different NAT rules via `iptables` on the router to cause traffic to be sent via the phone. An example of traffic forwarding, after applying NAT rules, is shown in Figure 6.

The NAT rules cause traffic to be sent to a specific port on the smartphone. Our Java program runs on the phone, binds to the specified port, and receives packets. It performs simplistic packet inspection and then sends it back to the router on a specific port. The router uses its NAT rewriting rules to send it on to the server. When a reply from the server is received by the router, the traffic likewise traverses the phone for inspection before traveling to the client.

We use the same three metrics as in the baseline and on-router cases to explore the performance characteristics of this phone-based inspection approach. In addition, we consider the CPU usage of the phone application itself, since high usage may result in battery depletion on the phone and could prevent its practical deployment.

Using the same settings as in the two prior sections, we explore the throughput when traffic is directed through the Moto G Power smartphone. In the middle two lines of Figures 8, we see that the median download throughput is 94.80 Mbps and the median upload throughput is 70.10 Mbps, with tight distributions (standard deviation of 4.32 Mbps for download and 2.87 Mbps for upload). The throughput is substantially higher than the on-router inspection approach in both Figure 8 and Figure 9. In effect, the processing of the NAT rules on the router may incur less computational overhead than the full process of inspecting the traffic. Since the router's CPU was the bottleneck in the on-router inspection scenario, this adjustment increases the rate at which traffic can flow.

In Figure 10, we confirm that the NAT-based approach yields significantly lower CPU utilization than on-device inspection when throughput is limited to 10 Mbps. The middle line in that graph shows that the NAT approach has a median of 24.0% CPU utilization with a standard deviation of 2.61%.

The insertion of another device on the network path, through a loop will necessarily increase the packet's propagation delay and may be observable in the overall end-to-end RTT. This is apparent

in Figure 11, with the RTT of the NAT approach represented by the two right-most lines. We see patterns where 20% of traffic has an RTT less than 30.44 ms, while 75% of traffic has an RTT over 120.17 ms. This is significantly higher than either the baseline scenario or when on-router inspection occurs. Importantly, this experiment uses a simple echo server approach and does not tax the CPU of the router. The on-router scenario would incur greater RTT delays when the CPU is a bottleneck due to processing delay.

In Figure 13, we explore the cause of the RTT delay in greater detail. We host a simple TCP echo program in three different ways. The left-most line represents the scenario in which the echo server runs on the server using the baseline approach (i.e., the traffic traverses the router to the server, bypassing the phone). The middle line represents the case in which the echo server runs in an application within an Android emulator running on a laptop. The two rightmost lines represent the echo server running on two separate physical smartphones: a Moto G Power and a Pixel 2. While the first two scenarios have fairly tight distributions, with RTTs less than 10 ms, the echo server built on the Moto G Power has a latency of around 20 ms for most traffic, and much longer delays for around 20% of traffic. Moreover, the echo server built on the Pixel 2 has a latency of less than 50 ms for around half of the trials, but has delays of over 200 ms for around half of the traffic. In essence, the simple echo server smartphone application sometimes incurs significant delay in sending or receiving traffic. While this occurs only around 20% of the time for the echo server, the proxy application would incur two instances of this behavior, causing more traffic to incur a delay.

The distinct RTT behaviors exhibited by the two physical phones, which are not present in the Android emulator, may indicate some outside effect due to phone-specific factors. These could include the use of power savings modes, in which applications are periodically suspended or queued to reduce energy consumption.

Our last metric explores the energy usage of the proxy application on the phone. We again use the Moto G Power smartphone as a proxy while maximizing throughput transmission from the client to the server. In this experiment, we also run a music-playing application on the phone, in the background, for comparison. We then record the CPU usage of the proxy application and the music application for 1,000 seconds using the `top` tool in the phone. We monitor the idle percentage of the 8 cores in the proxy device. In Table 4, we show the CPU usage of the proxy application and the music application, along with the time for which the CPU core is idle. In this table, 100% represents the full utilization of a single core on the device and 800% represents the full utilization of all eight device cores. The first row in Table 4 represents the proxy application, which uses only about 21% of a single core verses the roughly 107% CPU usage of the music application in the median case. We see that the majority of the device's computational resources are unused. Even in lower-end smartphones, the CPU impact of the proxy was about 20% of a single core. Therefore, we anticipate that the CPU-based energy

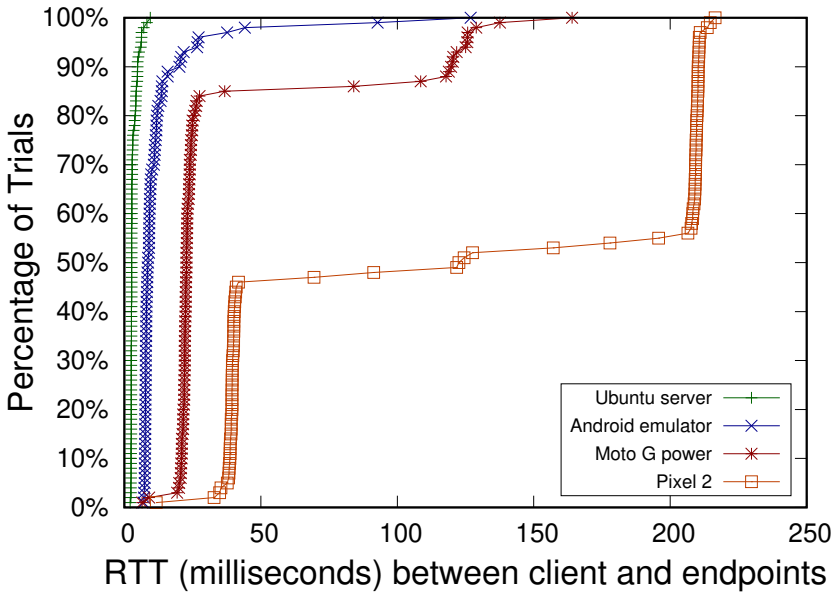


Figure 13: Comparison of RTT when connecting the client directly to the Ubuntu server, the Android emulator, the Moto G Power, and the Pixel 2.

consumption of the device would be a small fraction of a music application. Since phones are regularly used for music playing without significant power-related disruptions to end-users, it is likely that the proxy application would likewise represent a reasonable workload on phones.

Table 4: CPU usage of the smartphone for different applications when maximizing throughput while applying on-phone inspection.

Percentile of Trials	10th	50th	90th
CPU Usage of Proxy App	18%	21%	24%
CPU Usage of Music App	98%	107%	114%
CPU Idle	535%	560%	584%

4.3 Summary of Section

While residential networks need traffic inspection and analysis tools to protect their traffic, existing residential routers lack the computational resources for on-router inspection. Even a straightforward, IP address-based inspection tool on such a router can greatly limit the throughput the router can support. However, with carefully-crafted NAT rules, a router can redirect communication through another device, such as a smartphone, to inspect traffic. This opportunistic outsourcing of inspection to in-network devices avoids the privacy concerns associated with outsourcing such services to cloud providers.

In our experiments, we find that NAT-based diversion through a smartphone can substantially raise

the communication throughput from around 10 Mbps in an on-router implementation to around 90 Mbps through a smartphone. The router can periodically examine its ARP and DHCP data structures to detect the availability of a phone in the LAN, contact an application on the phone to configure proxy services, and then divert traffic through the phone to enable outsourced inspection. With such an approach, residential networks can opportunistically use available smartphones as middleboxes to enable higher-throughput traffic inspection.

5 Improving and Evaluating Appjudicator

Beyond utilizing Android OS to secure IoT devices and inspect home network traffic, we explore the use of Appjudicator [26] to secure network traffic through popular applications on Android devices. As mentioned in the section 2, Appjudicator is an Android application that works as a local VPN device and SDN agent. When it elevates network packets to a SDN controller, it also sends out the UI actions recorded by Android `AccessibilityServices` [32]. Therefore, it is built with a specific capability of fusing UI interactions and network activities. While Appjudicator is a good foundation for securing Android devices, it has several problems when implemented and requires comprehensive evaluations of its effectiveness and performance.

When initially using Appjudicator, we found several unsolved issues in its system. The Appjudicator uses Android's built-in API `VpnService` [6] to intercept local network traffic without root privilege. When it works as a local VPN that forwards network traffic from local applications to a remote server, it faces several challenges in performance and stability. There are TCP connections that are already closed between the client and the VPN service but are still connected between the VPN service and the remote server. When we compare the TCP connections on both sides of VPN service, we find that it generates more acknowledge packets in the TCP connection on the client's side than in the one on the server's side. Moreover, it sometimes fails in synchronization phase when building connections for the two sides. Therefore, at the start of our research, we do significant work in reviewing Appjudicator's running logs and fixing issues in the code.

We first revise part of the code to close the TCP connection.. Then we create a matching mechanism that compares every transforming TCP packet on both sides of the VPN service and detects any packets that do not match one on the other side. This mechanism helps us to fix the issue of having a large number of acknowledge packets. We use the debugger in Android Studio to find several programming conflicts and solve some suitability issues. While the Appjudicator is not a perfect Android application, we improve it to a version that can execute the experiments and conduct our research. In this Section, we aim to explore and answer the following two research questions:

- To what extent can the system that fuses UI interactions and network activities be used to

predict network flows on Android devices?

- To what extent can this on-device system influence the CPU, memory, battery, and network latency on mobile devices?

5.1 Evaluating Effectiveness of Appjudicator

In this Section, we aim to assess the role of the UI context of Appjudicator in network profiling. We investigate whether it can help identify if a network flow is due to an end-user action or an automated process, and how useful it is for network traffic monitoring. To achieve this, we follow a similar evaluation methodology as Chuluundorj *et al.* [4]. Specifically, we compare ourselves against traditional network-based sensors used in enterprise networks, which can be categorized into three broad classes:

- **IP Header Sensor**: This sensor only considers a network flow as a match if it has the same remote system's IP addresses and port numbers as in the idle or training data.
- **DNS-aware Sensor**: This sensor incorporates recent DNS queries with the IP Header Sensor data. In addition to matched network flows reported by the IP sensor, it considers a network flow as a match if it has the same host name as in the idle or training data.
- **UI-aware Sensor**: The UI sensor includes the matches from the DNS sensor as a baseline. The UI sensor would further consider a network flow as a match if the remote server appears in the UI context data.

We start our exploration by analyzing the background activity associated with an application without any end-user activity. To do so, we record all the network activities for three different sensor types while each tested application is idle. All such network traffic is added to an `Idle Period` allow-list that approves background activities that do not need user intervention.

Next, we train our research set-up to establish an association between end-user actions and network activity. We use Appium to automatically invoke a scripted sequence of UI elements. Instead of traversing all possible UI elements, we focus on different types of UI events in the Android accessibility service, such as clicks, text selections, and touch interactions. These events only occur when a user interacts with the device. In this training phase, we record the UI events and network behaviors observed at our sensors. We label this data as the `Training Data`, which records what UI information and network activity appears together in a non-compromised application. During our later testing phase, each sensor constructs allow-list rules using the training data to determine if it can correctly classify traffic.

App	Idle Period Samples	Training Samples	Testing Samples	Sensor Accuracy		
				IP Sensor	DNS Sensor	UI Sensor
Termius	236	1587	1669	6.6%	6.9%	99.1%
Chrome	208	1658	1811	4.8%	8.7%	98.6%
Firefox	280	1428	1506	5.3%	9.6%	99.4%
Gmail	180	1052	1313	43.6%	98.8%	100.0%
YouTube	315	1162	1845	57.3%	100.0%	100.0%

Table 5: Match rates across sensor types in the testing data set. DNS sensors appear to be more effective than IP sensors, but still have low match rates when destinations are specified by the user. The UI sensor can detect these destinations and leverage them in match rules.

For our evaluation, we selected popular applications from the Google Play Store that are widely used in people's daily lives, including Gmail, YouTube, Chrome, Firefox, and Termius (an SSH client), covering a broad range of usage scenarios. We note that Gmail and YouTube interact with a limited set of remote systems such as the servers associated with the application developer or advertisers. Chrome and Firefox allow users to specify arbitrary destinations, providing significant flexibility but making profiling efforts more complex. We also included Termius, an SSH client application that enables users to connect to user-specified destination servers. However, due to the large number of possible destinations, the training phase for these applications may be less effective, as it is unlikely that all future user-supplied destinations will be captured during training. Nonetheless, the UI sensor we used can provide useful information containing URLs, IP addresses, and DNS host names, which can help improve profiling efforts. In this experiment, we focused our analysis on user inputs in web browsers and SSH clients.

During the training phase when the application is idle, we run each application without user input for two hours to collect background network activity, which is added to an allow-list called the `Idle Period Samples`. For the `Training Data` phase, we create thirty workflow scripts for Gmail and YouTube, while for Chrome, Firefox, and Termius, we create workflow scripts that randomly visit websites from the top 500 websites list from `SimilarWeb` [30]. We collect data at the SDN controller for both datasets. In the testing phase, we execute the same workflow scripts while the SDN controller employs three rule matching sensors with rules constructed from the previous training data.

Table 5 presents the effectiveness results of our study, focusing on the accuracy of different sensor classes for various applications. For Gmail and YouTube, we observe that the Network Header Sensor has the lowest accuracy, likely due to DNS load balancing or the impact of content distribution networks (CDNs). On the other hand, the DNS sensor achieves an accuracy of over 98.8% for these applications, indicating the importance of host name matching rules.

When it comes to web browsers and SSH clients, the Network Header and DNS sensors exhibit lower accuracy rates, matching only 6.9% to 9.6% of traffic. However, our UI sensor achieves a match rate of over 98.6% at the controller, highlighting the significance of understanding user-specified destinations in traffic profiling. By integrating UI interactions and network activity, the UI sensor can effectively

perform network profiling with high accuracy.

5.2 Evaluating Performance of Appjudicator

We explore the performance implications of Appjudicator via an Android emulator consisting of a Pixel 4 with API level 30. We evaluate the performance in three metrics: networking latency, web traffic performance, and computational resources.

5.2.1 Networking Latency

Since our local VPN service intercepts all outgoing and incoming packets as they traverse the VPN service, it necessarily adds networking latency. To examine how much delay Appjudicator adds, we record the overall end-to-end delay between an application running in the Android emulator and a remote server on a public network. We perform the experiments on a simulated Google Pixel 4 phone with Android API level 30 and an Ubuntu 20.04 server. We run our SDN controller on a separate virtual machine on the same local network as the Android emulator. We use the remote server to periodically send packets with tailored payload to the application on the emulator and have the application simply echo it back. We compare the scenarios when Appjudicator is enabled and disabled. Since the variable in these cases is on the emulator side, we introduce a timer on the remote server side to record the overall end-to-end delay.

We measure the overall end-to-end delay by recording two timestamps at the remote server. The first timestamp is the time when a packet with a specific payload is sent from the remote server (t_1). The second is the time when the remote server receives the packet with same payload that the application echos back (t_2). When Appjudicator is enabled, the delay between these two timestamps includes any delays resulting from the consultation with the SDN agent and the VPN service. Accordingly, we consider the difference between t_2 and t_1 as the overall end-to-end delay between the application on the emulator and the remote server. To compare between Appjudicator being enabled and disabled, we can evaluate the total additional delay added by Appjudicator.

In our experiments, we collect the timestamps over 1,000 trials. As shown by the blue line in Figure 14, the average overall end-to-end delay of packets with Appjudicator disabled is 20.37ms, which represents the networking latency between our local network to the remote server. The green line shows the overall end-to-end delay with Appjudicator enabled, and its average delay of packets is 24.97ms. For transferring network packets between the application on the emulator and on the remote server on a public network, Appjudicator adds 4.60 ms, on average to the latency.

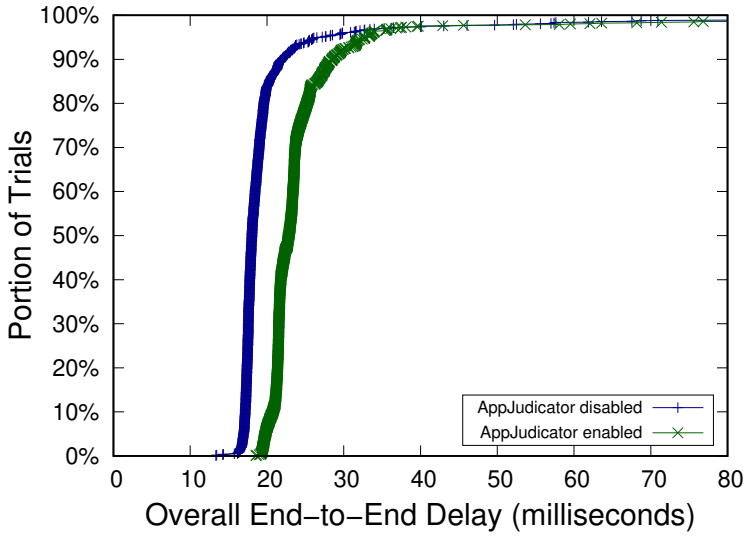


Figure 14: Comparison of overall end-to-end delay when APPJUDICATOR is enabled and disabled.

5.2.2 Web Traffic Performance

Beyond measuring the delay added to an established TCP flow, we want to examine how much latency Appjudicator adds when the user tries to access a new website. The first packet of each new outgoing flow needs to wait for context from the UI Monitor, elevation to the SDN controller, and installation of the rule by the local Android SDN agent. We explore the web page load time to evaluate the delay caused by these operations.

We conduct these experiments in the same Android emulator and network configuration as in the previous evaluation. Our SDN controller runs on the same local network as the emulator. We use the `Firefox DevTools` to record the web page load time in the Firefox web browser in the emulator. In the cases where Appjudicator is enabled and disabled, we interact with the browser to access the Top 50 Websites listed on `SimilarWeb` [30] and collect the web page load time in Figure 15.

The results show that, when we use the browser without Appjudicator running, the average page load time for Top 50 websites is 453.4 ms. After we enable Appjudicator, the average page load time is increased to 1333.9 ms. We note that Appjudicator would add about 880 ms latency to a first-time page access for the websites that have not been recorded by our SDN controller.

5.2.3 Computational Resources

Energy consumption is a significant consideration for mobile devices. Accordingly, we evaluate battery consumption, memory usage and CPU usage of Appjudicator on the Android emulator. Our tests use Android Studio's `Profiler` [13] and `top` tool to gather data. We run a Google Pixel 4 with an Android

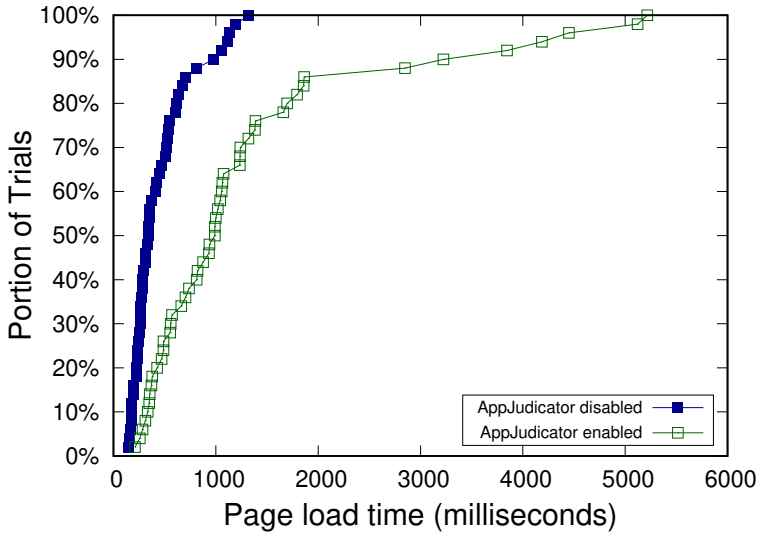


Figure 15: Comparison of web page load time between Appjudicator enabled and disabled

API level of 30 on an Android emulator with 4 cores and 2GB of memory.

During the experiments, we first start Appjudicator, and then play music in YouTube Music to emulate a typical user's usage of the device. We use the monitoring tools to record the CPU usage of Appjudicator and YouTube Music. The monitoring tools record a data point for each second, and the duration of the experiment is 1,000 seconds.

As shown in Figure 16 and Table 6, when Appjudicator runs on the virtual Android device, it consumes a relatively constant amount of memory, with an average of 128.2 MBytes. It uses less memory than the YouTube Music application, which uses an average of 300.2 MBytes memory. The CPU consumption of Appjudicator averages around 11.9%, which is higher than the average CPU usage by YouTube Music at 3.7%. Moreover, the Android Studio's `Profiler` classifies the energy consumption level of the application as "light." As a result, we do not believe that Appjudicator has a substantial impact on computational resources.

Table 6: Comparison of Memory (MEM) usage between Appjudicator and YouTube Music

Percentile of Trials	10th	50th	90th
MEM Usage of Appjudicator (MBytes)	127.0	129.0	131.0
MEM Usage of YouTube Music (MBytes)	291.8	299.7	309.7

5.3 Summary of Section

In this Section, we implement and evaluate Appjudicator, an SDN architecture for mobile devices that associates UI elements with network flows. With the ability to consult external SDN controllers for

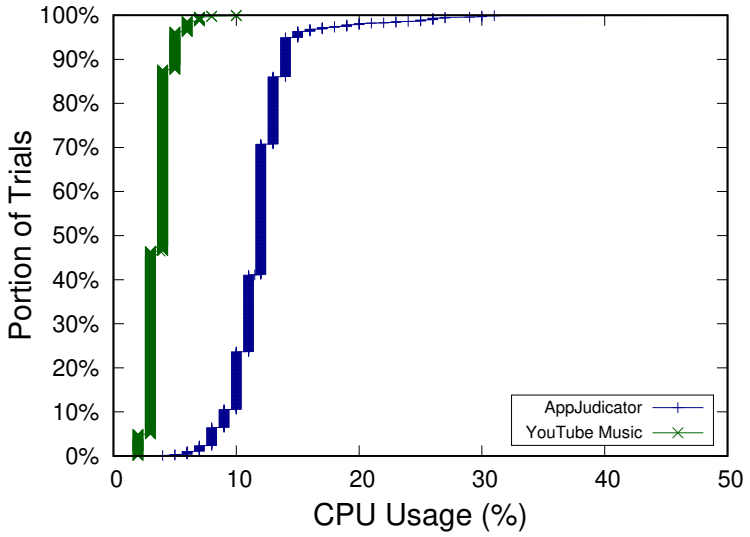


Figure 16: Comparison of CPU usage between Appjudicator and YouTube Music

assistance, the Appjudicator tool can respond to evolving threats while providing sufficient context for access control decisions. This mechanism helps increase the accuracy of network profiling from 9.6% to 98.6%. In our evaluation, we find that Appjudicator adds 4.60 ms on average to the network latency and consumes an average of 128.2 MBytes memory of Android emulator. We note it consumes acceptable computation resources while introducing modest network delay. We consider Appjudicator to be a useful and practical tool for Android OS to secure its local applications.

6 Future Work and Conclusion

In this thesis, we explore ways to leverage Android OS to help secure various devices in smart home networks. We first introduce fusing UI interactions and network flows to secure IoT devices. By utilizing state machines to model IoT device actions and enforcing dynamic allow-list policies, we are able to effectively control access to IoT devices and prevent anomalous traffic. Our system only allows traffic that is known to be legitimate, making it more difficult for adversaries to launch effective attacks. Attackers can only communicate with an IoT device after a legitimate user interaction and must conform to known legitimate interactions in terms of packet sizes and timing. Any traffic that does not match a legitimate pattern will be filtered, triggering an alert. Our experiments demonstrate that this approach is effective across different manufacturers, device types, and communication protocols. Although each new device requires a training phase, the resulting policy can be used across instances of the same device. This means that manufacturers or security service providers can create policies and distribute them across SDN controllers to protect a large number of IoT devices. Future work

could expand this approach to include additional types of IoT devices, such as media streamers or smart speakers, and different types of longer-range connectivity.

We also explore the use of opportunistically outsourced middleboxes for traffic inspection in home networks. Existing residential routers do not have sufficient computational resources for on-router inspection, making it challenging to protect residential network traffic. Even a simple IP address-based inspection tool on such a router can significantly reduce the router's throughput. However, by using carefully-crafted NAT rules, a router can redirect communication through another device in the network, such as a smartphone, to inspect traffic. This approach avoids privacy concerns associated with outsourcing inspection services to cloud providers. Our experiments indicate that NAT-based diversion through a smartphone can substantially increase communication throughput from around 10 Mbps with on-router implementation to approximately 90 Mbps using a smartphone. To implement this approach, the router periodically examines its ARP and DHCP data structures to determine the availability of a phone in the LAN. If a phone is available, the router contacts an application on the phone to configure proxy services and divert traffic through the phone to enable outsourced inspection. By leveraging available smartphones as middleboxes, residential networks can opportunistically improve the throughput of traffic inspection. We also do significant work to improve and evaluate Appjudicator. Our experiments in evaluating its effectiveness and performance show that it effectively secures the traffic from applications on the Android smartphone. Moreover, when we implement it on an Android emulator, it ensures the overall network performance and does not consume major computing resources.

In conclusion, as the number of connected devices in smart home networks continues to increase, the need for securing diverse devices has grown significantly. With the lack of powerful security tools and trained personnel that are available in enterprise networks, securing these devices poses a nontrivial challenge. This work proposes several methods to utilize the Android smartphone operating system to secure various devices in home networks. By exploring access control mechanisms and leveraging existing available devices, this software-based solution offers an efficient and cost-effective way to inspect and secure network traffic. The results of the experiments demonstrate the effectiveness of the proposed approach, highlighting its potential for securing home network devices and protecting the privacy of users.

References

[1] Zahrah A Almusaylim and Noor Zaman. A review on smart home present state and challenges: linked to context-awareness internet of things (iot). *Wireless networks*, 25(6):3193–3204, 2019.

[2] 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.

[3] Zorigtbaatar Chuluundorj, Shuwen Liu, and Craig A Shue. Generating stateful policies for iot device security with cross-device sensors. In *2022 13th International Conference on Network of the Future (NoF)*, pages 1–9. IEEE, 2022.

[4] Zorigtbaatar Chuluundorj, Curtis R Taylor, Robert J Walls, and Craig A Shue. Can the user help? leveraging user actions for network profiling. In *2021 Eighth International Conference on Software Defined Systems (SDS)*, pages 1–8. IEEE, 2021.

[5] Soteris Demetriou, Nan Zhang, Yeonjoon Lee, XiaoFeng Wang, Carl A Gunter, Xiaoyong Zhou, and Michael Grace. Hanguard: Sdn-driven protection of smart home wifi devices from malicious mobile apps. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 122–133, 2017.

[6] Android Developers. Vpnservice. https://developer.android.com/reference/kotlin/android/net/VpnService, 2022.

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

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

[9] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/, 2020.

[10] Nick Feamster. Outsourcing home network security. In *ACM SIGCOMM Workshop on Home Networks*, pages 37–42. ACM, 2010.

[11] Julien Gedeon, Christian Meurisch, Disha Bhat, Michael Stein, Lin Wang, and Max Mühlhäuser. Router-based brokering for surrogate discovery in edge computing. In *International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 145–150. IEEE, 2017.

[12] 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.

[13] Google Developers. Measure app performance with Android Profiler. https://developer.android.com/studio/profile/android-profiler, October 2020.

[14] Ibbad Hafeez, Markku Antikainen, Aaron Yi Ding, and Sasu Tarkoma. Iot-keeper: Securing iot communications in edge networks. *arXiv preprint arXiv:1810.08415*, 2018.

[15] Hall, Michael, and Raj Jain. Performance analysis of openvpn on a consumer grade router. *cse. wustl. edu*, 2008.

[16] Sungmin Hong, Robert Baykov, Lei Xu, Srinath Nadimpalli, and Guofei Gu. Towards sdn-defined programmable byod (bring your own device) security. In *NDSS*, 2016.

[17] 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.

[18] Steve TK Jan, Qingying Hao, Tianrui Hu, Jiameng Pu, Sonal Oswal, Gang Wang, and Bimal Viswanath. Throwing darts in the dark? detecting bots with limited data using neural data augmentation. In *IEEE Symposium on Security and Privacy (IEEE SP)*, 2020.

[19] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[20] Jeongkeun Lee, Mostafa Uddin, Jean Tourrilhes, Souvik Sen, Sujata Banerjee, Manfred Arndt, Kyu-Han Kim, and Tamer Nadeem. mesdn: Mobile extension of sdn. In *Proceedings of the fifth international workshop on Mobile cloud computing & services*, pages 7–14, 2014.

[21] Yunsen Lei, Julian P Lanson, Remy M Kaldawy, Jeffrey Estrada, and Craig A Shue. Can host-based sdns rival the traffic engineering abilities of switch-based sdns? In *2020 11th International Conference on Network of the Future (NoF)*, pages 91–99. IEEE, 2020.

[22] Yunsen Lei and Craig A Shue. Detecting root-level endpoint sensor compromises with correlated activity. In *International Conference on Security and Privacy in Communication Systems*, pages 273–286. Springer, 2019.

[23] 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.

[24] Rwan Mahmoud, Tasneem Yousuf, Fadi Aloul, and Imran Zualkernan. Internet of things (IoT) security: Current status, challenges and prospective measures. In *International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 336–341. IEEE, 2015.

[25] 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.

[26] Joseph Petitti. *Appjudicator: Enhancing Android Network Analysis through UI Monitoring.* PhD thesis, WORCESTER POLYTECHNIC INSTITUTE, 2021.

[27] 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 *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.

[28] Camille L Ryan and Jamie M Lewis. *Computer and internet use in the United States: 2016.* US Department of Commerce, Economics and Statistics Administration, US . . . , 2017.

[29] 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.

[30] SimilarWeb. Top websites ranking. https://www.similarweb.com/top-websites/, 2023.

[31] 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.

[32] Android Studio. Create your own accessibility service. https://developer.android.com/guide/topics/ui/accessibility/service, 2019.

[33] Android Studio. Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey, 2020.

[34] Yulei Sui, Yifei Zhang, Wei Zheng, Manqing Zhang, and Jingling Xue. Event trace reduction for effective bug replay of android apps via differential gui state analysis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1095–1099, 2019.

[35] 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.

[36] Curtis R Taylor, Douglas C MacFarland, Doran R Smestad, and Craig A Shue. Contextual, flow-based access control with scalable host-based sdn techniques. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[37] Curtis R Taylor and Craig A Shue. Validating security protocols with cloud-based middleboxes. In *IEEE Conference on Communications and Network Security (CNS)*, pages 261–269. IEEE, 2016.

[38] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D Breaux, and Jianwei Niu. Guileak: Tracing privacy policy claims on user input data for android applications. In *Proceedings of the 40th International Conference on Software Engineering*, pages 37–47, 2018.

[39] 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.

[40] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054, 2013.

[41] 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.

[42] Eric Zeng, Shrirang Mare, and Franziska Roesner. End user security and privacy concerns with smart homes. In *Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 65–80, 2017.

[43] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104, 2012.