Appjudicator:
Enhancing Android Network Analysis through UI Monitoring

by

Joseph Petitti

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

____

May 2021

APPROVED:

____

Professor Craig A. Shue, Major Thesis Advisor

____

Professor Robert J. Walls, Thesis Reader

____

Professor Craig E. Wills, Head of Department

Abstract

Smartphones are becoming increasingly important in all aspects of life, including corporate environments, where "bring your own device" (BYOD) policies are gaining widespread acceptance. Malware already exists to take advantage of Android phones in BYOD settings, aiming to take control of devices with access to privileged information by disguising itself as a benign app. Malware could be easier to detect if network administrators had more insight into employee-owned smartphones. We propose a system, called APPJUDICATOR, to address this issue. It implements an accessibility service to monitor user interactions with the user interface (UI) of other apps, so this context can be used in malware detection. For example, if an app sends a new network request without any user interaction, this flow could be the result of malware and should be investigated. Our app is a host-based software defined networking (SDN) agent that works in conjunction with an SDN controller to monitor and control the phone's networking abilities based on the organization's SDN rules and our UI context. We build a proof of concept application and find that it can successfully combine network and UI data while adding less than 14 milliseconds of total latency in 95% of flows.

Acknowledgments

I would like to express my gratitude to my advisor, Professor Craig Shue, for guiding me through my graduate education with helpful advice and insightful feedback. Without his patience and wisdom this thesis could not have happened. I would also like to thank Yu Liu for his technical help and support throughout the project, along with Matt Puentes and Cole Granof for their advice and moral support. I also wish to thank Beckley Schowalter for her help with writing and grammar. Additionally, I want to express my appreciation to my thesis reader, Professor Robert Walls, for his helpful feedback on this project. His questions and criticism, along with those from every member of the WPI Cake Lab, are what turned this paper from a very rough summary to a polished final product.

Contents

1 Introduction

Many employees are now using their own personal smartphones for work purposes, as working from home becomes widespread and companies adopt "Bring Your Own Device" (BYOD) policies. Existing corporate network administration tools have little insight into the connections made by employee-owned smartphones. It is difficult for large organizations to maintain security across their networks with such little knowledge about, and control over, these mobile devices. The fact that these devices enter and exit the corporate network and connect to other networks, such as their cellular data provider's, on a daily basis complicates this issue.

Smartphones have a wide range of powerful networking and computational abilities, and are typically privately owned by employees, which raises further concerns when integrating them into secure network environments. For example, smartphones have been targeted by malware specifically designed to infiltrate corporate networks [20], such as "Dresscode", which disguises itself as a legitimate app in order to steal data and add infected devices to a botnet [31]. The "xHelper" malware targets phones used for both personal and professional purposes; can automatically download and install arbitrary software specified by an attacker; and persists even after a factory reset [37]. A black market malware-as-a-service model called "Black Rose Lucy" even offers control of infected Android devices to paying customers, potentially giving any malicious actor an entry point to a secure network [38].

If mobile devices are to have access to sensitive corporate network infrastructure, either from home or from work, administrators need new tools to monitor and control their network connections. These tools could provide the information needed to monitor network activity for malware, provision network resources based on which apps are being used, or debug complex distributed applications. However, if the tools are not easy to deploy and manage, it will be difficult to convince end users to adopt them. For example, similar solutions on Windows [3, 23] require administrator privileges or the ability to modify the kernel. This is impractical for most users to do on a smartphone and exposes them to additional security risks [10].

To solve these issues on Android devices, we propose a new app called Appjudicator that leverages user interface (UI) interaction and software-defined networking (SDN) principles to determine whether network flows are legitimately user-initiated. The app has two components: an accessibility service that monitors UI interaction and a VPN service that intercepts network flows. We use these two sources of information to correlate network flows with user interaction, and we provide this information to network administrators by acting as a host-based SDN agent.

The UI monitoring component uses Android's accessibility API to asynchronously record physical hardware inputs, such as tapping or swiping the touch screen. This gives administrators the ability to accurately distinguish between human-driven and automated network requests. Meanwhile, the VPN service acts as an OpenFlow agent and rule cache, giving administrators fine-grained control over which connections the device is allowed to make. Flows are augmented with context from the specific device and application, and can be elevated to the organization's SDN controller along with UI context data if necessary.

Our application provides organizations with a new set of powerful, easy-to-use tools for monitoring BYOD Android smartphones in a simple app package that is easy for users to install. It does not require rooting, recompiling the kernel, or any other cumbersome processes. It can distinguish between user-generated and automated network requests with a high degree of confidence using techniques that are difficult for malware to evade. This makes users safer by detecting stealthy malware on their devices and improves the organization's network security.

The costs of our system include the resource overhead of running the app (CPU cycles, battery power, etc.) and the expense of installing, running, and maintaining the controller server. We found these costs to be a relatively small addition to existing resources. The app's UI monitoring and VPN service also add some latency to network requests performed on the phone. In our experiments, Appjudicator added less than 14 milliseconds of total latency to 95% of packets processed by the app.

Using UI data to distinguish user-initiated behavior and using a smartphone as an SDN agent have previously been studied, but Appjudicator combines these aspects in new and important ways. Android host-based SDN agets have been investigated by Hanguard [5], and using UI interaction as context for identifying malicious app behavior was examined in AppIntent [41], but our system combines these approaches to form a novel solution for a distinct use case. One key difference is that AppIntent uses machine learning to conduct static analysis of apps, while Appjudicator enables live monitoring and response to malicious network flows in real time.

Kwon et al. have proposed a system that uses UI data to distinguish user-generated network flows, and the combination of UI monitoring and SDN has been implemented on Microsoft Windows by Harbinger [3]. We solve new challenges by implementing this strategy on the Android platform. For example, since we are constrained by Android's rigid permissions system, we do not have root access to the device and cannot modify the kernel. These restrictions require novel techniques and systems.

This work focuses on implementing a proof of concept networking tool for Android. We explore the following research question:

> *Can UI interaction and network activity successfully be used to predict and associate network flows with user actions on Android devices with acceptable overhead?*

We leave the potential applications of this data for network security to future work, instead examining the effectiveness of our strategy for differentiating user-initiated flows and measuring the system's CPU, memory, and energy costs.

To investigate, we perform a study with several real apps to determine whether APPJUDICATOR can successfully detect user-initiated network requests. We found that even with a simple timing-based strategy the app could successfully associate network flows with the UI interaction that initiated them. We also measure the total added latency of the VPN and SDN agent, finding that our system adds less than 14 milliseconds of total end-to-end latency to 95% of intercepted packets. On a standard 60 Hz screen, this latency is imperceptible to users.

In summary, this paper makes the following contributions:

- We propose a novel system for associating network flows with user interface context on Android. Unlike existing solutions, our application implements a host-based SDN agent and does not require root access to the device.

- We design and implement a practical prototype of the system on Android, called APPJUDICATOR, which integrates with other SDN infrastructure using a subset of the OpenFlow protocol.

- We evaluate the performance and efficacy of APPJUDICATOR on real and virtual Android devices with real-world apps. The results show that this prototype has a low impact on CPU, memory, and battery usage, and adds a minimal amount of total latency to network communications while achieving its goal of differentiating user-initiated flows and associating flows with UI context.

2 Background and Related Work

Related work has been conducted in both of the main focus areas of this project: the use of UI interaction data for security and the use of Android smartphones in software-defined networking (SDN). We now review such work.

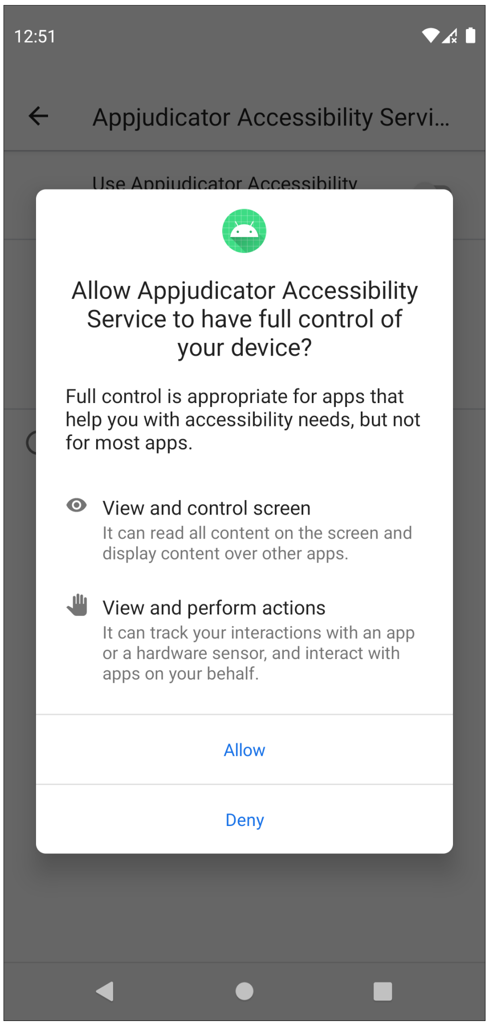
2.1 UI Interaction and Security

Previous work has investigated how user interactions with the UI can be used to identify human-initiated behavior on various platforms. Harbinger examines user interface activity on Windows to provide context for network requests to previously unknown hosts in a default-deny environment [3]. This approach "hooks into" mouse and keyboard inputs using Microsoft's UI Automation library [26], intercepting the input before it is received by the intended application. Although Harbinger's operations are performed synchronously, they still added only six milliseconds or less of latency to 96% of flows.[1]

Kwon et al. also used UI interaction data to distinguish human-generated from automated network requests on Windows [23]. They propose a host-based system for combating botnets by leveraging UI interaction data. But their approach of labelling any flow initiated within one second of an interaction with the flow's process as "user generated" ignores much of the context that can be gained from UI data.

Shirley and Evans attempt to infer a users' intentions from their actions and to define a language for writing access control policies based on these intentions [33]. Their approach collects UI interactions like mouse clicks and keystrokes, along with the state of the user interface. Like other previous work mentioned above, they focus on Windows only, and use their program to limit malware file access.

Cui et al. proposed a system called BINDER for Windows that attempts to detect malicious network flows by associating user input events, to process start and finish events, and network events [4]. Their system would only block flows locally and was never implemented. In contrast, our system integrates into an SDN environment and provides context to network administrators.


12:51
Appjudicator Accessibility Servi...
Use Appjudicator Accessibility

Allow Appjudicator Accessibility Service to have full control of your device?

Full control is appropriate for apps that help you with accessibility needs, but not for most apps.

View and control screen
It can read all content on the screen and display content over other apps.

View and perform actions
It can track your interactions with an app or a hardware sensor, and interact with apps on your behalf.

Allow

Deny

Figure 1: Android displays this warning when enabling an accessibility service.

2.2 Android's Accessibility Service

Most previous work on Android has used the platform's `AccessibilityService`, a library that includes the ability to respond to UI changes among other accessibility features [13]. Apps that implement this API can be notified asynchronously of UI state transitions in other all apps on the phone.

These services break the operating system's normally strong sandboxing principles by their ability to read and interact with anything displayed on the screen [19, 6]. Real Android malware has taken advantage of accessibility services to spy on users and take control of mobile devices [22]. For security reasons, Android requires any accessibility service to be enabled manually by the user in the settings app. The operating system displays a dialog box to warn the user of the potential security risks involved with accessibility services when one is first enabled (see Figure 1). Still, some have argued that even this

[1]In contrast, the relevant Android library is asynchronous and event-driven [13]. Section 3.2 discusses the benefits and challenges associated with this approach.

warning is not descriptive enough of the powerful and wide-reaching permissions given to accessibility services [19].

Some prior work has been done on detecting malicious behavior using an accessibility service. AppIntent [41], for example, uses this service to determine if an app is leaking private information by building a graph of UI interactions that could lead to information leakage. However, AppIntent is only designed to determine whether an app *could* leak private information through static analysis and does not work in real time on a per-flow basis.

2.3 Software-defined Networking

Software-defined networking (SDN) has been studied as a tool for giving network administrators more information about, and control over, intra-network traffic. In essence, SDN centralizes network intelligence by separating the forwarding of packets (the data plane) from their routing (the control plane). In this paradigm, network switches merely forward packets, while all control and logic is centralized with an SDN controller server [21].

The most commonly used SDN protocol is OpenFlow, a standard maintained by the Open Networking Foundation [8]. The OpenFlow protocol provides a standard way for SDN agents (usually network switches) to cache packet-forwarding rules [30]. When a packet from a new flow arrives at the agent, it looks up the most specific rule that applies to the packet, then performs the actions listed in the rule. These actions can include changing fields in the packet's header, forwarding it out a specific port, or dropping the flow entirely. If the switch has no matching rule in its flow table—or if a rule specifically requests it—the switch can forward the packet in question to the central controller server to ask what to do with it.

SDN gives administrators more fine-grained control over packet routing rules, which enables more sophisticated policy enforcement. It also provides insight into potential issues and congestion in the network from one centralized control point, rather than from many switches and routers. The main drawback of this approach is the high overhead that causes it to scale poorly [1].

Several techniques have been tried for distributing the load on the controller server [29, 7]. Some authors have investigated using end nodes, instead of routers and switches, as SDN rule caches to alleviate this problem [36, 3]. In this approach, each host caches rules from the SDN controller and makes routing decisions about its own network flows. APPJUDICATOR follows this strategy, implementing a host-based SDN agent and extending the OpenFlow protocol with more context.

Taylor et al. explore the feasibility of separating the SDN controller from the local network entirely, connecting instead to a cloud-based controller [35]. They achieved acceptable overhead for 90% of end users by selecting low-latency public cloud locations.

Others have investigated the potential for improving SDN's capabilities by adding context [40], an approach which APPJUDICATOR also attempts. SDN rules could be more powerful and granular with more metadata, such as a network flow's initiating application.

Qazi et al. attempt to classify traffic from Android phones using machine learning on an SDN switch [32]. Their approach achieves 94% accuracy in the top 40 Android applications. By utilizing the phone itself as an SDN agent, APPJUDICATOR can identify a network flow's originating application with 100% accuracy for all apps.

2.4 The Android Phone as an SDN Agent

Several previous applications have used an Android smartphone as an SDN agent and rule cache. Hong et al. explore the tools available to organizations for managing BYOD Android devices and propose a system for applying SDN policies to these phones [18]. They apply app-specific rules that are aware of device context, such as GPS location, with low overhead. However, this work does not take advantage of the many extra sources of context the mobile device can provide.

While Hong et al. [18] and APPJUDICATOR are both aimed at corporate users, HanGuard investigates the use of SDN principles to protect Internet of things (IoT) devices in home networks [5]. Their application provides tools for users to easily define SDN rules for IoT apps, IoT devices, and users.

While each of these areas has been extensively researched alone, only Chuluundorj [3] and Kwon et al. [23] have investigated how UI and network data can be combined for security purposes. However, both of these applications were developed for the Windows platform—bringing the idea to Android involves a new set of constraints and challenges. For example, our proposed solution can be installed as a normal app, and does not require recompiling the kernel or rooting the phone. This means that we cannot modify the operating system kernel as the Windows solutions do, and must abide by Android's strict permissions system.

3 Approach and Implementation

Appjudicator works by analyzing network flows with the added context of UI interaction data. It uses this information as part of a host-based software-defined networking (SDN) agent to make decisions about whether to allow or block individual flows. The app is composed of three primary components: (1) a VPN service that captures and analyzes network flows from the device, (2) an accessibility service that monitors user interactions with the UI, and (3) an SDN agent that implements a subset of the OpenFlow 1.0 specification [30].

The app is implemented using Kotlin, an object-oriented programming language that is interoperable with Java and compiles to Java Virtual Machine bytecode. This is Google's preferred language for Android development, and makes some tasks like null-checking and concurrency easier than they would be in Java [24]. As a proof of concept, the app's user interface is minimal, simply providing a list of which services are running and a method to toggle them (see Figure 2).

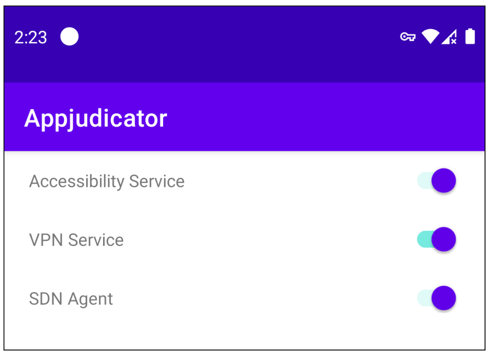


Figure 2: A screenshot of Appjudicator's user interface (vertical whitespace truncated for brevity).

The VPN service implements Android's `VpnService` API [15], but connects to a packet-capturing class running on the same phone rather than a remote server. The UI-monitoring component implements Android's `AccessibilityService` API [13], and the SDN agent implements a subset of the OpenFlow 1.0 switch standard [25]. Each of these components work together to correlate network flows with a UI interaction (if any) that initiated them, and to elevate suspicious flows to the SDN controller. We now describe each of these components and explain how they work together.

3.1 VPN Service

The networking component of Appjudicator utilizes Android's built-in API for redirecting network traffic, the `VpnService` [15]. However, rather than connecting to a remote VPN server, we connect to a simple server running locally on the device. We register a new VPN connection using this API and prompt the user to connect to it. This routes all the device's traffic through our service, which captures and logs packets before forwarding them along to their destination.

Once activated, the VPN service runs in the background and receives packets from the operating system through a `ParcelFileDescriptor` instance that can be used to read and write packets from the network interface's buffer [12]. Normally, a VPN service would forward packets through a tunnel to a remote VPN server, but Appjudicator's VPN server runs locally on the device. Packets are instead passed to processes running on other threads that log the packets and forward them along to their destination using Java sockets.

It is normally the app's responsibility to encrypt data being transferred to the VPN gateway [12], but this is unnecessary in Appjudicator because both client and server are running on the same device. We benefit from the reduced computational resource consumption and energy use that result from not having to encrypt and decrypt packets.

3.1.1 Deconstructing Flows

Captured packets are parsed, logged, and reconstructed by the VPN service using Pcap4J, a third-party Java IP packet library [39]. Source and destination IP addresses are collected from the packet's IP header, along with source and destination ports from the TCP or UDP header. Appjudicator currently only supports IPv4, so any intercepted IPv6 packets are simply dropped. Likewise, packets with unknown transport-layer protocols (*i.e.*, neither TCP nor UDP) are dropped. The network context provided by the VPN service includes IP source and destination addresses, protocol, source and destination ports, payload size, initiating application, and full packet payload, all of which can be used to enforce fine-grained SDN rules.

Appjudicator associates an app with the flows it created. On Android API versions Q and later, we can use the `ConnectivityManager` to query the operating system for which user ID owns a particular flow. Because each app has a different user ID in Android, we can then use the `packageManager` to look up the package name for that UID as long as we requested the QUERY_ALL_PACKAGES permission. See Listing 1 for an example of how to do this in Kotlin.

Listing 1: Source code to obtain the package that created a given network flow.

```kotlin
@RequiresApi(Build.VERSION_CODES.Q)
fun connToPackage(
    protocol: Int, // either TCP (6) or UDP (17)
    localAddress: InetSocketAddress,
    remoteAddress: InetSocketAddress,
    context: Context
): String {
        val cm = context.getSystemService(Context.CONNECTIVITY_SERVICE)
                        as ConnectivityManager
        val uid = cm.getConnectionOwnerUid(
                protocol,
                localAddress,
                remoteAddress
        )
        if (uid == android.os.Process.INVALID_UID) return "unknown"
        return context.packageManager.getNameForUid(uid) ?: "unknown"
}
```

Packets are organized by flow, essentially a communication channel between one application and another. A flow is defined as a sequence of packets with the same source IP address, source port, destination IP address, destination port, and transport layer protocol (either TCP or UDP). Packets are stored in queues by flow while awaiting a response from the SDN agent. These queues are stored in a hash map, indexed by the concatenation of the flow's addresses, ports, and protocol. When the SDN agent reaches a decision, the VPN service looks up the flow and, based on the agent's decision, it either forwards all queued packets in order or drops them.

3.1.2 Multithreading

Network connections are performance-critical, so we need to ensure that the user experience is not blocked waiting for Appjudicator to process network requests. The VPN service creates new IO threads using Kotlin's concurrency framework so that network processing can be performed off the main thread. This allows the VPN service to have a minimal impact on performance and prevents the UI from blocking while waiting for network operations.

3.1.3 Permissions and Security

For the VPN service to work, Appjudicator must request the INTERNET permission and declare a service that requests the BIND_VPN_SERVICE permission in the Android manifest. It must also name a route from which to capture traffic. We use 0.0.0.0/0 to capture all IPv4 traffic, but this could be changed so the VPN is only used on traffic of a particular interface or subnet.

After the app is installed, the VPN service can be started or stopped directly from it. Appjudicator provides simple buttons to do this, or it can be configured to connect to the VPN automatically on startup. The service can also be stopped or disabled from Android's Settings menu for security reasons.

There is one kind of traffic we never want to block: communications between the SDN agent and controller. The SDN agent uses the VpnService.protect() method [15] on the socket channel to the controller to make sure this traffic is not intercepted by the VPN.

3.2 Accessibility Service

Android's AccessibilityService API assists individuals who need tools like screen readers and automated UI navigators. By registering an accessibility service with the operating system, we can be notified of changes in the UI state of other applications. These changes are referred to as accessibility events, and are asynchronously triggered by the Android operating system and delivered to the listening accessibility service. By using this API, Appjudicator's UI monitoring component can be informed of almost every use interface interaction in any every app on the device.

Accessibility services can be very dangerous from a security perspective, so they must be registered with the operating system and enabled manually. In Section 3.2.4 we provide more details on how to do so programmatically.

3.2.1 Accessibility Events

An accessibility event represents a single state change in the user interface of an app, such as a button being pressed, a view being swiped, or the focus changing [11]. An accessibility service can specify the particular app packages and event types it wants to receive, but Appjudicator registers to receive all event types from all packages. The accessibility event is fired by a view (the basic building block of Android user interfaces) and passed to interested parties by the operating system [16].

The accessibility event object contains some context about the UI interaction, including the type of UI element, the type of interaction (*e.g.*, click, swipe, long press), and descriptive text of the element [11]. Additional information about the UI element that initiated the event can be retrieved with the `AccessibilityEvent.getSource()` method. This method allows the app to get information about the layout hierarchy, providing context about the element's enclosing views and child elements. Because this context information could potentially expose private user data, the service must declare the `canRetrieveWindowContent` attribute in its configuration XML file. With this attribute set, Android will warn the user that the application can retrieve the contents of the screen when it is enabled.

By default, the operating system only includes view objects it thinks are important to accessibility with the accessibility event, but we can request information about all views instead by passing the `FLAG_INCLUDE_NOT_IMPORTANT_VIEWS` flag to the accessibility service [11].

These accessibility events are triggered for almost every type of UI interaction, including clicks, swipes, long presses, and even input from devices like virtual or physical keyboard.

However, events fired from apps that do not use Android's UI libraries do not provide as much context. For example, we cannot get layout hierarchy information from an app that renders its UI in OpenGL or some other graphics platform [11].

3.2.2 Asynchronous Event Handling

When an accessibility event is generated, the Android operating system passes the generated object to the accessibility service's `onAccessibilityEvent()` callback asynchronously [13]. The event handler does not block the user interface because it runs on a different thread, which helps achieve our goal of adding minimal latency. However, this asynchronous processing also means an app may continue generating accessibility events while an earlier event is still being processed, so we must make sure to handle events efficiently.

There is theoretically a race condition if a `packet_in` message[2] is ready to be sent before the most recent UI interaction is finished processing. In practice, however, queuing packets, performing a SDN flow table lookup, and preparing a `packet_in` message takes longer than the single hash map lookup and linked list insertion the accessibility service performs.

3.2.3 Identifying UI Elements

Accessibility events do not provide a unique identifier for UI elements, so we implement a system inspired by Fazzini et al. [9]. Android UI elements, called views, may have an ID, but these are not guaranteed to be unique or even present on every view. Note that for the operating system to report view IDs in accessibility nodes, we need to pass the `FLAG_REPORT_VIEW_IDS` flag to the accessibility service. We use the initiating element's resource ID and resort to a selector based on the element's position in the XML UI tree if the ID is missing or non-unique. This allows us to precisely correlate a network flow with the particular UI element that initiated it. These selectors should also be relatively stable across multiple application launches, because they will not change as long as the app's UI structure remains constant. Developers rarely make large structural changes to app interfaces to follow common human-computer interaction guidelines [27].

[2]A `packet_in` message is how the app notifies the company's SDN controller that it received a flow and would like instructions on what to do with it. SDN is explained in more detail in Section 3.3.2.

3.2.4 Permissions and Security

Appjudicator declares a service that requests the `BIND_ACCESSIBILITY_SERVICE` permission in the Android manifest file. This declaration tells Android which class represents the service and specifies another XML configuration file. The file provides metadata about the service, such as listing which types of accessibility events to subscribe to, and describes the service's purpose to be shown to the user when it is enabled.

For security reasons, an app cannot enable its own accessibility service [19]. To prevent malware from taking advantage of the far-reaching permissions of accessibility services, a user must manually enable the service in the system settings after being prompted with a dialogue box that explains some of the risks involved (see Figure 1). Appjudicator can only point the user toward the Settings page; the rest must be done manually. A user may have any number of accessibility services running at a time, but they must be individually manually enabled and each display a persistent notification reminding the user that the service is still running in the background.

3.3 Host-Based SDN

Appjudicator is designed to integrate with corporate SDN infrastructure, so it needs to be able to communicate with an SDN controller server. The OpenFlow switch specification, the de facto standard for software-defined networking, describes a protocol for switch-controller communication [30]. This specification is designed for large network switches connected to potentially dozens of hosts, so implementing it on a smartphone in Kotlin requires some special considerations.[3]

Large portions of the OpenFlow switch specification simply do not apply to Appjudicator. For example, the app has no concept of Ethernet frames and only one "physical port" in the sense that it is a switch for only one host. Appjudicator does not support any optional features of the specification. It can initiate a connection with an OpenFlow controller, report its supported features, process `flow_mod` messages, send `packet_in` messages, and receive `packet_out` messages. We provide more information on the OpenFlow specification in Section 3.3.2.

3.3.1 SDN Rule Cache

Like any SDN agent, Appjudicator maintains a cache of rules to apply to network flows that pass through it. Rules can match any combination of source or destination IP addresses, source or destination ports, and protocol, or have wildcards for any of those fields. Each rule contains instructions on what types of flows to match and a list of actions to perform on matched flows. We support only the minimum allowed set of possible flow actions: forward and drop.

When a new connection is created, the VPN service notifies the SDN agent and queues all packets from that flow until it gets a response from the SDN agent. The SDN agent looks up the flow in its flow table. It finds the most specific matching rule (*i.e.*, the matching rule that used the fewest wildcards) and sends its actions back to the VPN service. If no matching rule is found in the flow table, the agent elevates the flow to the controller along with UI context in a `packet_in` message. Figure 3 illustrates this process. Section 3.3.2 provides more information about `packet_in` and `packet_out` messages.

Appjudicator will continue forwarding, blocking, or elevating packets in other flows while waiting for a response from the SDN controller. The controller makes a decision based on the network and UI context and sends back a `packet_out` response describing what to do with the flow.

3.3.2 OpenFlow Protocol

The OpenFlow Switch Specification describes a protocol for SDN agents to connect to a controller and exchange messages [30]. Appjudicator implements the minimum subset of the OpenFlow Protocol (OFP). All OpenFlow messages are sent over TCP in OpenFlow packets. These packets have a simple 8-byte header that describes the OpenFlow version, the type of the message, the total length of the packet, and the transaction ID (see Figure 4). Replies to an OpenFlow message have the same transaction ID as the request.

When the app is launched, it opens a new TCP connection to a user-configurable IP address or host name that runs the controller server. The controller and switch each send a `hello` message to initiate the connection. The controller then queries the switch to see what capabilities and features it supports. Appjudicator replies with a `switch_features` message explaining that it has one physical port (the

[3]Sections 2.3 and 2.4 describe prior work in host-based SDN agents on Android and other platforms.

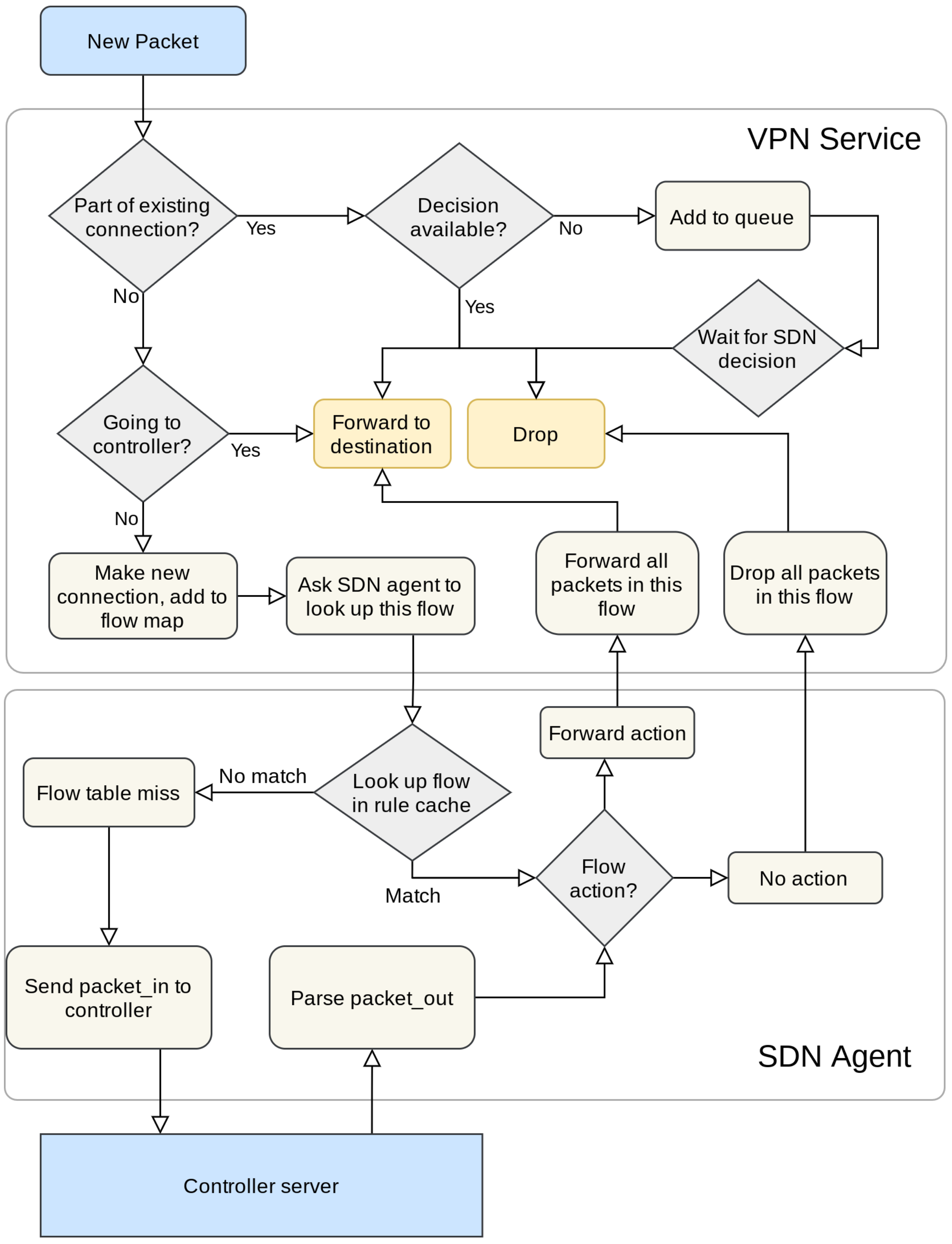


Figure 3: Flow chart of the steps used to decide whether to forward or drop a packet.

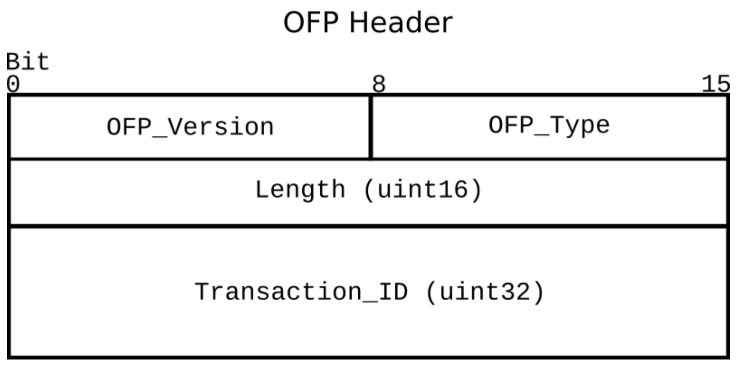


Figure 4: Fields of an OpenFlow packet header.

phone itself) and only supports two packet actions: forward and drop. At this point the connection is fully established, and the controller and agent can both send messages back and forth. Figure 5 depicts this process.

While the app is running, the controller can modify flow rules on the switch by sending a `flow_mod` message. These instruct the switch to add or remove one or more flow rules, and they are the primary way to enforce new policy decisions.

The final type of communication between the SDN agent and controller is `packet_in`/`packet_out` messages. When the SDN agent receives a packet that does not match any flow rules, the agent must ask the controller what to do with the packet. We construct a customized version of the standard OFP `packet_in` message to send to the controller, including UI interaction metadata.[4] Figure 6 shows the fields included in these messages. The first eight bytes are the standard OFP header. The "buffer ID" value will be included in the corresponding response so the relevant packet can be identified. The "in port" field does not apply to host-based SDN agents like Appjudicator. The reason field tells the controller whether this `packet_in` is being sent because of a flow table miss or because an action explicitly requested sending the packet to the controller.

Appjudicator has no concept of link layer protocols, so the field for the Ethernet header is zeroed out. Following this are the full contents of the IP packet in question. Up to this point, the `packet_in` follows the OpenFlow specification exactly, but after the packet contents we fill the rest of the space available (up to 1500 bytes) with UI context, including UI events from the same app that initiated the connection, their types, sources, and UI hierarchy. IP packets sent to the controller are usually the first packet of a flow,[5] which are normally short, so there is often enough space to include the most relevant UI events.

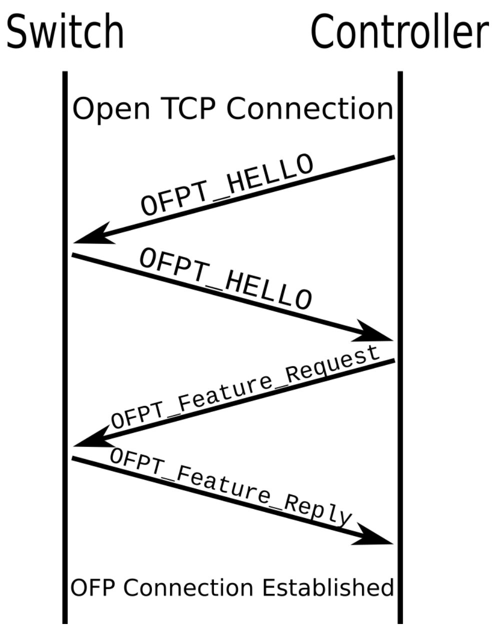


Figure 5: OpenFlow handshake diagram.

The controller responds with a standard `packet_out` message, which tells the agent which actions to take on the flow. Optionally, the agent can remember this response by adding a new flow table rule that matches the same flow and applies the same actions, so the controller will not have to be queried again in the future. This is not part of the OpenFlow specification, but is a feature of Appjudicator.

3.3.3 Associating Network Flows with Context

When the SDN agent intercepts a new flow to a previously unknown domain, it tries to associate it with UI context. The app looks up any accessibility events from the past two seconds that originated from the same app that owns the network connection (see Section 3.1.1). We found that a time interval of two

[4]Section 3.2 describes this metadata and how it is obtained.
[5]Such as a TCP SYN packet.

seconds is long enough to catch most user-initiated network requests without also associating unrelated UI interactions.

If there is no likely initiator of the network flow within the two prior seconds, the flow is considered non-user-initiated. The SDN agent includes information about these UI events with the packet (including the event type and view hierarchy[6]) as it is elevated to the controller. This UI context should allow for more powerful and fine-grained SDN rules.

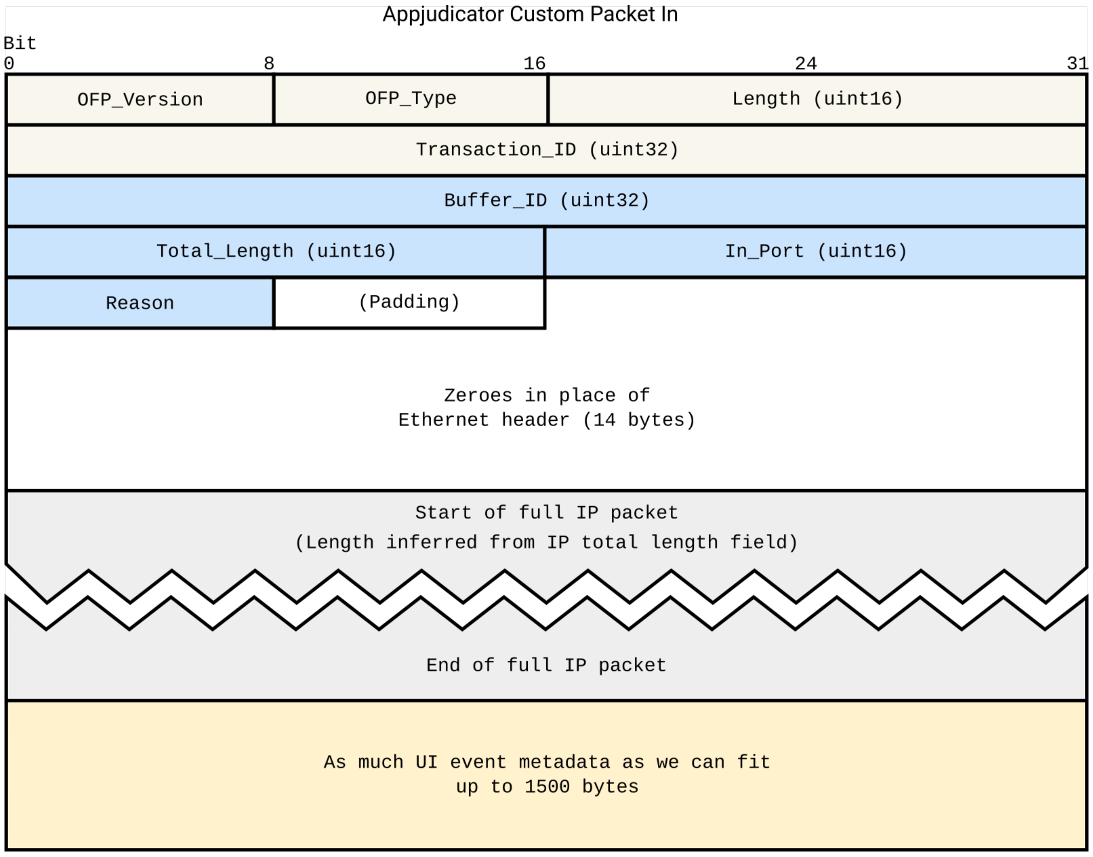

Appjudicator Custom Packet In
Bit
0
8
16
24
31
OFP_Version
OFP_Type
Length (uint16)
Transaction_ID (uint32)
Buffer_ID (uint32)
Total_Length (uint16)
In_Port (uint16)
Reason
(Padding)
Zeroes in place of Ethernet header (14 bytes)
Start of full IP packet (Length inferred from IP total length field)
End of full IP packet
As much UI event metadata as we can fit up to 1500 bytes

Figure 6: Fields in APPJUDICATOR's custom `packet_in` message.

3.3.4 Default-Allow Flows

Our system must account for non-user-initiated flows that occur as part of the normal operation of the device. To do this, we profile the Android operating system and common applications in advance. While running in calibration mode, APPJUDICATOR adds the IP address and ports of all non-user-initiated requests to a default-allow map. Then later, while operating normally, the app will not question flows to these pairs of IP address and ports even if they are not user-initiated.

[6]Section 3.2.3 explains how we construct a view hierarchy in more detail.

4 Evaluation and Results

We seek to evaluate whether Appjudicator achieves its goal of distinguishing user-initiated network flows using UI context and whether it does so with acceptable overhead. For this performance evaluation, we perform a practical analysis and measure the app's resource cost. Following are the procedures used for each of these experiments and their results.

4.1 Differentiating User-generated Flows

The goal of this experiment is to determine whether Appjudicator can successfully distinguish between network flows that were specifically initiated by a human user and flows generated automatically by an app or the Android system. We attempt to justify the parameters used for defining a flow as "user initiated."

4.1.1 Experimental Setup

This test was performed on a virtual Google Pixel 4, with API level 30 (the newest available). We perform actions on the phone to simulate real life use cases with Appjudicator, including browsing the web in Chrome, exploring an app store (F-Droid), and playing a game with advertisements. We use our app's logging to record whether each flow is considered "user initiated," along with the timing of UI interactions and flow table lookups.

To evaluate the effectiveness of our UI element system for identifying UI elements (described in Section 3.2.3), we record the proportion of accessibility events in our sampling period that came from sources with a unique resource ID.

4.1.2 Results

Appjudicator was successful in identifying 100% of new flows that occurred within two seconds of a UI interaction as user initiated. This raises the question of how effective this two-second cutoff is. Based on our experience, two seconds seems to be generous enough to allow all flows associated with the UI interaction to be triggered, without being so lenient that unrelated flows are also allowed.

Professional sources generally support this reasoning. Google's developer blog recommends two seconds of total page load time as "the threshold for ecommerce site 'acceptability'" [28]. Search engine optimization firm Semrush writes, "Serve your customers with the page load time they need, a good goal being 1–2 seconds" [2].

Figure 7 illustrates this cutoff. It shows four typical page loads in Chrome, with each point in the plot representing a new flow made by Chrome. The accessibility event that initiated the page load (clicking on a link) occurs at time 0. The dashed gray line shows the two-second cutoff. All the flows that occurred before this cutoff, within the green shaded area, are considered to be user initiated, while flows that occurred after the cutoff are not. These flows are likely the result of asynchronous JavaScript network calls initiated by the page, not directly by user action.

In our trial, we found that approximately 85% of accessibility events had a source element with a unique resource ID. Even dynamically-generated UI elements often have resource IDs. For example, links in webpages are given IDs generated by Chrome. This is good news for our system, because resource IDs are the easiest and most effective way to uniquely identify UI elements from accessibility events. Still, this figure may vary in apps in which developers do not give every element an ID.

4.2 Performance Evaluation

To fulfill its purpose as an always-enabled enterprise network security tool, Appjudicator has to be as unobtrusive to end users as possible. This is especially important because the app performs some actions on every network flow, so any latency added by it would be especially noticeable.

4.2.1 Measuring Added Latency

To measure the effect of Appjudicator on network communication speed, we measure the total additional latency added by the app. We measure added latency in all incoming and outgoing packets, but focus on the first packet of each new flow in particular, because these require the most processing time.

For this test, we run the app in an Android virtual machine simulating a Google Pixel 4, on API level 30. The controller server is a simple Python script that responds to every `packet_in` message with

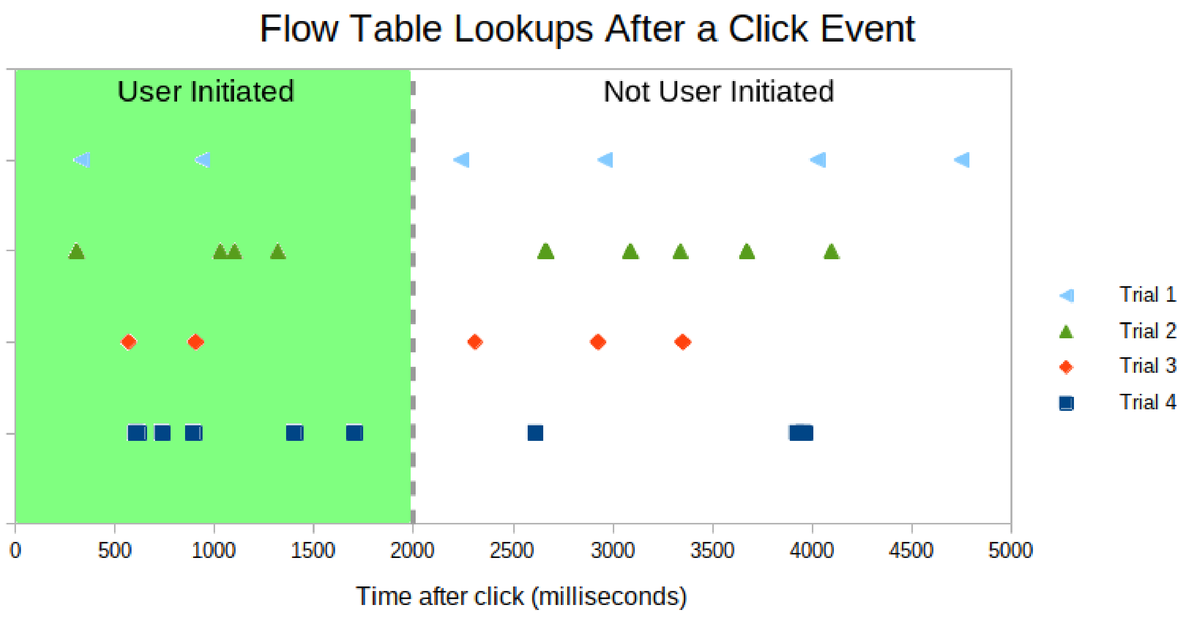


Figure 7: The delay between a click event and the new flows that it initiates. The dashed vertical line shows the two-second cutoff: flows before this line (in the green area) are considered to be associated with the click event.

a `packet_out` message instructing the agent to forward the flow.[7] This controller is run on a separate machine on the same local network as the phone. The average round-trip ping time between the phone and the controller server was 0.950 milliseconds.

In the test, the SDN agent starts with an empty table of flow rules, so it must send a `packet_in` to the controller for every new flow. After the agent gets a response from the controller, all subsequent packets in the same flow follow the action specified by the response. Section 3.3.2 describes the OpenFlow protocol in more detail.

We focus particularly on the added latency of the first packet in a new flow, because this is where the system does the most processing. The start of a new flow requires an SDN flow table lookup and possibly communication with the SDN controller. Creating a new TCP or UDP connection requires allocating memory for queued packets and metadata, and then inserting this data in a lookup table. Subsequent packets in the same flow will require less processing overhead.

We configure the app to log a timestamp when a packet is first intercepted by the VPN. Appjudicator's VPN service parses the packet and determines whether it is part of an existing connection or not, and adds it to a queue. If it is the start of a new connection, the VPN service queries the SDN agent and continues processing other packets while waiting for a response. The SDN agent performs a flow table lookup, and sends a `packet_in` to the SDN controller if it fails to find a matching rule. If this is the case, it waits for a response from the controller, then performs a `flow_mod` operation to cache the result as a new rule. Finally the SDN agent instructs the VPN service to drop or forward the flow. In this test, all flows are allowed, so the VPN service removes the packet from its queue and writes it to the correct network interface file descriptor.[8] After all of Appjudicator's processing is finished, a second timestamp is logged. The earlier timestamp is the time the packet would have been sent to the network without our system's interference, and the second timestamp is the time it actually was sent. The difference is the total latency added by the app. We also log the overall round-trip time of each flow, from when the first packet is sent to when its response is received, so we can compare this to the time added by Appjudicator. All timestamps are created using Android's `SystemClock.elapsedRealtimeNanos()` method, which returns the number of nanoseconds since the system was booted [17].

[7]This test is designed to measure Appjudicator's efficiency only. In a production environment, a more complex controller server would likely add a non-negligible amount of delay while it processes a `packet_in`.

[8]Figure 3 illustrates this process.

4.2.2 Latency Test Results

There were 1,162 packet timing traces logged during the sampling period, of which four were removed as outliers, leaving 1,158 data points. The average added latency among all packets was 6.17 milliseconds, with a standard deviation of 6.20 milliseconds.

During the sampling period, 212 timing traces were logged for the first packet of new flows, of which one was removed as an outlier, leaving 212 data points. These packets had higher added latency than others, and their added latency varied more. The average added latency among these packets was 12.12 milliseconds, with a standard deviation of 7.36 milliseconds. Figure 8 charts the full results of this test.

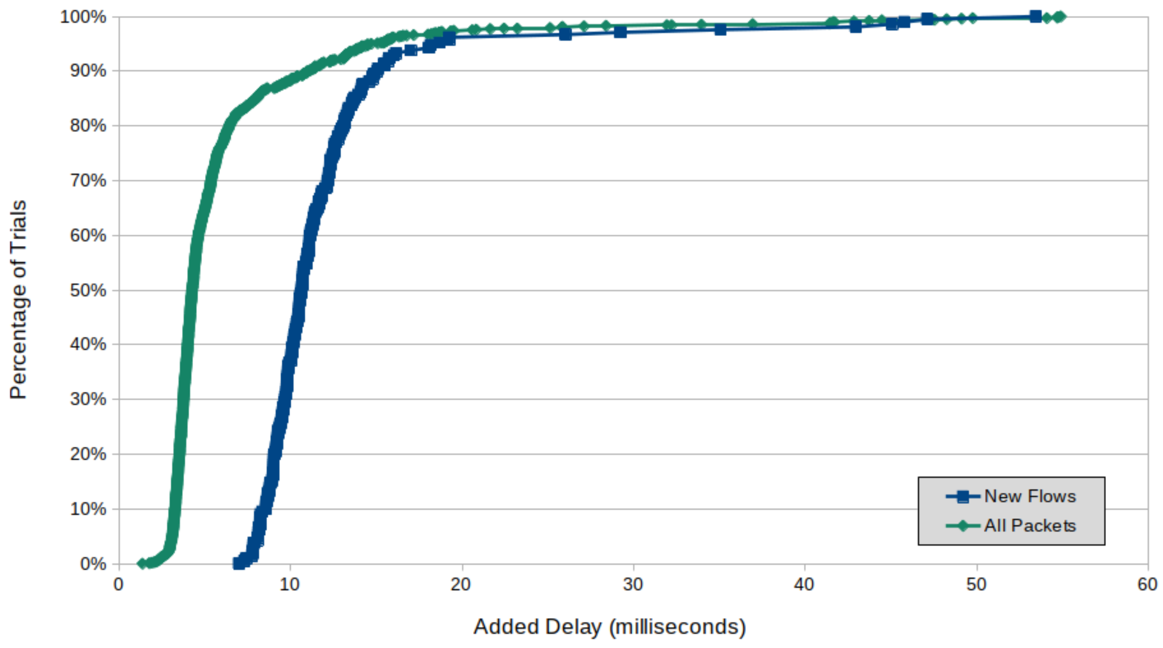


Figure 8: Total latency added by Appjudicator

Traces which recorded a total added latency greater than 150 milliseconds were considered outliers. This is more than 12 times the new flow average latency, and over 24 times the average latency of all packets. Because these outliers are so rare, making up less than 0.5% of all packets, they may have been caused by poor network conditions.

In our sample, 95% of packets were processed with less than 14 milliseconds of total added delay. On a 60 Hz display, one frame lasts 16 milliseconds, so any delay less than this is imperceptible to users. In our sampling period, the latency added by our system was usually small compared to the total round-trip time (RTT) of a new flow. Figure 9 compares the delay added by Appjudicator with the total RTT of flows.

In measuring the timing of individual functions, we found that the initial VPN overhead (reading, processing, and queuing packets) took an average of 44 microseconds. Looking up flows in the flow table took an average of 53 microseconds with 10 initial rules, although this will increase as more flow table entries are added. Accessibility event lookup took an average of 63 microseconds. By far the biggest contributor to overall latency in our trial was sending `packet_in` messages and waiting for a response from the controller. The SDN agent waited 2963 microseconds on average for a `packet_out` message, which includes an average of 950 microseconds of network travel time between the phone and the controller server.

4.2.3 Measuring Resource Overhead

Appjudicator has costs in both added latency and increased resource consumption. Running the VPN service, accessibility service, and SDN agent in the background consumes additional processing cycles, memory, and battery life. We measure the resource overhead with Android Studio's Profiler [14].

This test was performed twice, first on an Android virtual machine simulating a Google Pixel 4, on API level 30, and second on a physical Google Pixel 3, also on API level 30. We ran the app with the

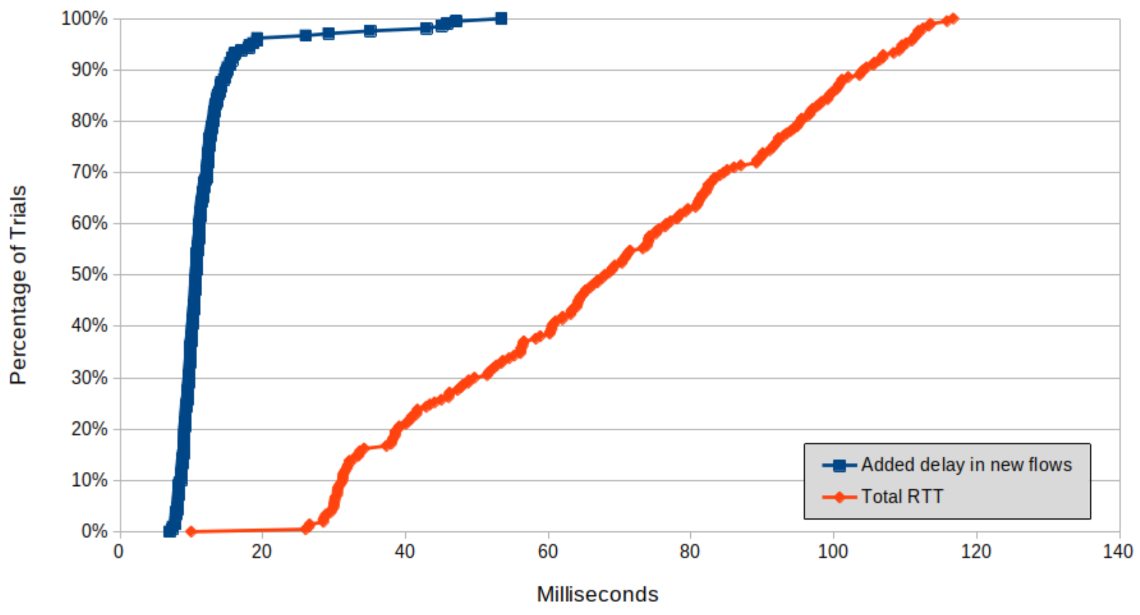


Figure 9: The delay added by our system even for new flows is a relatively small piece of the total network latency of these flows.

accessibility service, VPN service, and SDN agent all enabled while recording its CPU, memory, and energy usage with Android Studio's Profiler. Each test lasted 30 minutes, during which time we used various other apps including Chrome, Termux, and F-Droid while APPJUDICATOR ran in the background.

During the virtual trial, APPJUDICATOR's memory usage remained relatively constant. It peaked at at 150.1 MB, with an average of 137.8 MB. Its processor usage also usually remained low, averaging 5% of the virtual machine's CPU but occasionally spiking as high as 39%. The profiler categorized the app's energy usage as "light," because it used only minor CPU resources and did not use any location resources, wake locks, or alarms.

Results were similar in the physical trial. APPJUDICATOR used less memory on the physical device, peaking at 122.3 MB and averaging 85.4 MB. This may be because the Pixel 3 has less available RAM than the Pixel 4. CPU usage was higher on the physical phone, averaging 17% and spiking as high as 55%. Again this may be because of hardware differences, as the Pixel 3 has an older, slower processor. The app's energy usage was again classified as "light" by the profiler during the physical trial.

Based on these results, we conclude that APPJUDICATOR has a minor impact on CPU, memory, and battery usage on modern hardware, and therefore could be used effectively as an always-on background security application.

5 Discussion

Here we discuss the practical applications of our work, as well as its limitations, future work, and possible improvements. We also use the results of our work to answer our original research question.

5.1 Implications and Applications

Appjudicator was designed for an enterprise BYOD setting, as a tool to give system administrators more control over employee-owned smartphones. Our app would work with an OpenFlow SDN controller to enforce company policy rules. With some modification, Appjudicator could also be used in other applications. For example, our app could adapted for use by personal users in residential networks, connected to a cloud-based SDN controller like the one proposed by Taylor et al. [35]

The use of Appjudicator in an enterprise network could raise some privacy concerns for end users. Users may object to having logs of every UI interaction they make on their personal device sent to their employer. The system could potentially expose private data to the SDN controller, [19] so administrators will have to carefully consider how their policy rules impact user privacy. A potential solution to this could be to define a set of general user intent profiles on the device, and only send the closest matching profile to the server with a `packet_in` rather than the entire UI interaction metadata.

5.2 Future Work

More work is needed to implement Appjudicator as a practical network security tool in an enterprise setting. Our work's biggest limitation is that it is only available for Android. Android is the leading smartphone operating system, capturing over 71% of the global market share, [34] but implementing a similar system on iOS (which makes up virtually all of the remaining market share) would make the system accessible to all BYOD employees. This work would be necessary before a company could require its employees to use Appjudicator—extra insight and control over *only* Android users would provide little benefit.

Custom SDN controller software could also be written to take advantage of Appjudicator's enhanced context information. Our app sends metadata about recent UI interactions along with each `packet_in`, so a custom controller could take advantage of this extra context in its decision-making process. For example, a controller could have a more sophisticated algorithm for inferring user intent, and use this to block non-user-initiated flows from mobile devices. A custom controller could also empower system administrators to write fine-grained, context-aware SDN policies.

Further work could also be done to mitigate some of the limitations discussed in Section 5.3. Because our work is a proof of concept, little development time was spent on optimization or graphical polish. A more user-friendly UI and settings menu could be implemented for the app. Our latency test results (described in Section 4.2.2) indicate that the app adds less than 10 milliseconds of total latency to 95% of processed packets, but this could probably be reduced with further performance optimizations.

Some researchers have investigated SDN architectures that involve multiple distributed controller servers. [7, 29] Appjudicator currently connects to only one statically-defined controller server, but could be extended in future work to connect to several.

5.3 Limitations

For ease of development and testing, Appjudicator only supports IPv4, and can only handle UDP and TCP as transport layer protocols. This is enough to power the vast majority of everyday networking applications, but support could be added for IPv6 and other transport layer protocols.

Our strategy for defining user-generated flows can give a rough estimate of whether a user is interacting with a particular app, but it is far from perfect. A flow from an application is considered user-initiated if a user made any UI interaction with that app in the past two seconds, so malicious network flows could be allowed if they happen to be sent within this time period. A malicious app with knowledge of this system could also generate fake accessibility events to give the impression that a user is interacting with an app. To mitigate this issue, future work could focus on improving how we define a flow as user-initiated. For example, a system for detecting user intent similar to the one presented by Shirley and Evans [33] could be added to Appjudicator. A system like this would make it easier to detect whether whether the accessibility events we receive are legitimate.

5.4 Conclusion

In this thesis, we implemented a host-based SDN agent and UI monitoring system on Android. We demonstrated that these systems can be combined to collect network data and UI context, and this data can be used to successfully distinguish which flows on the device are legitimately user initiated. Our experiments showed that the system can run in the background with minimal overhead to CPU, memory, and battery usage, and adding an acceptably low amount of overall end-to-end latency to network operations on the device. Finally, we discussed how the application can be used as a network security tool in enterprise environments and what future work is required to turn our proof of concept into a practical application.

References

[1] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui, “Software-defined networking (sdn): a survey,” *Security and Communication Networks*, vol. 9, no. 18, pp. 5803–5833, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1737

[2] C. Bird, “What is a good page load time for SEO – how fast is fast enough?” Dec. 2020. [Online]. Available: https://www.semrush.com/blog/how-fast-is-fast-enough-page-load-time-and-your-bottom-line/

[3] Z. Chuluundorj, “Augmenting network flows with user interface context to inform access control decisions,” Master’s thesis, Worcester Polytechnic Institute, Dec. 2019. [Online]. Available: https://digitalcommons.wpi.edu/etd-theses/1331

[4] W. Cui, R. H. Katz, and W.-t. Tan, “Binder: An extrusion-based break-in detector for personal computers,” in *USENIX Annual Technical Conference, General Track*, 2005, pp. 363–366.

[5] S. Demetriou, N. Zhang, Y. Lee, X. Wang, C. A. Gunter, X. Zhou, and M. 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*, 2017, pp. 122–133.

[6] W. Diao, Y. Zhang, L. Zhang, Z. Li, F. Xu, X. Pan, X. Liu, J. Weng, K. Zhang, and X. Wang, “Kindness is a risky business: on the usage of the accessibility apis in android,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2019, pp. 261–275.

[7] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, “Towards an elastic distributed sdn controller,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, pp. 7–12.

[8] D. Erickson, “Open networking foundation formed to speed network innovation,” Mar. 2011. [Online]. Available: https://web.archive.org/web/20140116023421/http://archive.openflow.org/wp/2011/03/open-networking-foundation-formed-to-speed-network-innovation/

[9] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, “Barista: A technique for recording, encoding, and running platform independent android tests,” in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 149–160.

[10] Google, “Security risks with modified (rooted) Android versions,” 2020. [Online]. Available: https://support.google.com/accounts/answer/9211246?hl=en

[11] Google Developers, “Create your own accessibility service — Android developers,” Dec. 2019. [Online]. Available: https://developer.android.com/guide/topics/ui/accessibility/service

[12] ——, “VPN — Android developers,” Dec. 2019. [Online]. Available: https://developer.android.com/guide/topics/connectivity/vpn

[13] ——, “AccessibilityService — Android developers,” Sep. 2020. [Online]. Available: https://developer.android.com/reference/android/accessibilityservice/AccessibilityService

[14] ——, “Measure app performance with Android Profiler,” Oct. 2020. [Online]. Available: https://developer.android.com/studio/profile/android-profiler

[15] ——, “VpnService — Android developers,” Aug. 2020. [Online]. Available: https://developer.android.com/reference/kotlin/android/net/VpnService

[16] ——, “AccessibilityEvent — Android developer,” Feb. 2021. [Online]. Available: https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent

[17] ——, “Systemclock,” Feb. 2021. [Online]. Available: https://developer.android.com/reference/android/os/SystemClock

[18] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu, “Towards sdn-defined programmable byod (bring your own device) security,” in *NDSS*, Feb. 2016.

[19] A. Kalysch, D. Bove, and T. Müller, “How android’s ui security is undermined by accessibility,” in *Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium*, 2018, pp. 1–10.

[20] M. Kan, “Android malware that can infiltrate corporate networks is spreading,” 2016. [Online]. Available: https://www.computerworld.com/article/3126390/android-malware-that-can-infiltrate-corporate-networks-is-spreading.html

[21] H. Kim and N. Feamster, “Improving network management with software defined networking,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[22] J. Kraunelis, Y. Chen, Z. Ling, X. Fu, and W. Zhao, “On malware leveraging the android accessibility framework,” in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services.* Springer, 2013, pp. 512–523.

[23] J. Kwon, J. Lee, and H. Lee, “Hidden bot detection by tracing non-human generated traffic at the zombie host,” in *International Conference on Information Security Practice and Experience.* Springer, 2011, pp. 343–361.

[24] F. Lardinois, “Kotlin is now google’s preferred language for android app development,” May 2019. [Online]. Available: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/

[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: enabling innovation in campus networks,” *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[26] Microsoft, “UI automation - win32 apps — Microsoft docs,” May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/winauto/entry-uiauto-win32

[27] D. Norman, *The design of everyday things: Revised and expanded edition.* Basic books, 2013.

[28] M. Ohye, “You and site performance, sitting in a tree...” May 2010. [Online]. Available: https://developers.google.com/search/blog/2010/05/you-and-site-performance-sitting-in

[29] Y. E. Oktian, S. Lee, H. Lee, and J. Lam, “Distributed sdn controller system: A survey on design choice,” *computer networks*, vol. 121, pp. 100–111, 2017.

[30] Open Networking Foundation, “Openflow switch specification,” Open Networking Foundation, Tech. Rep., Dec. 2009. [Online]. Available: https://opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf

[31] D. Palmer, “Over 400 instances of dresscode malware found on google play store, say researchers,” Oct. 2016. [Online]. Available: https://www.zdnet.com/article/over-400-instances-of-dresscode-malware-found-on-google-play-store-say-researchers/

[32] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir, “Application-awareness in sdn,” in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, 2013, pp. 487–488.

[33] J. Shirley and D. Evans, “The user is not the enemy: Fighting malware by tracking user intentions,” in *Proceedings of the 2008 New Security Paradigms Workshop*, 2008, pp. 33–45.

[34] StatCounter, “Mobile operating system market share worldwide,” Mar. 2021. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide/2021

[35] C. R. Taylor, T. Guo, C. A. Shue, and M. E. Najd, “On the feasibility of cloud-based sdn controllers for residential networks,” in *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN).* IEEE, 2017, pp. 1–6.

[36] C. R. Taylor, “Software-defined networking: Improving security for enterprise and home networks,” Ph.D. dissertation, Worcester Polytechnic Institute, 2017. [Online]. Available: https://digitalcommons.wpi.edu/etd-dissertations/161/

[37] J. Vijayan, “‘Unkillable’ Android malware app continues to infect devices worldwide,” Apr. 2020. [Online]. Available: https://www.darkreading.com/mobile/unkillable-android-malware-app-continues-to-infect-devices-worldwide/d/d-id/1337519

[38] W. Wong, “New malware-as-a-service threat targets Android phones,” Sep. 2018. [Online]. Available: https://securityintelligence.com/news/new-malware-as-a-service-threat-targets-android-phones/

[39] K. Yamada, “Pcap4J: A Java library for capturing, crafting, and sending packets,” 2016. [Online]. Available: https://www.pcap4j.org/

[40] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, “AppContext: Differentiating malicious and benign mobile app behaviors using context,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 303–313.

[41] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, “AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1043–1054.