Privacy Monitor

A Major Qualifying Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfilment of the requirements for the

Degree of Bachelor of Science

in

Computer Science

by

Matthew Hlushko

Ivan Martinovic

Jacob Salerno

May 2023

APPROVED

Professor Craig A. Shue, MQP Advisor

Abstract

This project extends the capabilities of Appjudicator, an Android Packet Capturing Application which couples an Application's UI Interaction with outgoing network packets and forwards them to an off-device host-based SDN system. The PrivacyMonitor module incorporates safeguards and user-centric controls in Appjudicator that strike a practical balance between the user's interest in minimizing the potential privacy impact and maximizing Appjudicator's security benefits. We achieve this by allowing the user to define their privacy expectations, generalizing network data to preserve privacy, providing external control through policy updates, and adding support for Transport Layer Security inspection, or TLSi.

Contents

1 Introduction

Bring Your Own Device (BYOD) is a policy by which a company allows its employees to bring their own personal devices such as smartphones, tablets, or laptops to the workplace and use them for work-related purposes. The popularity of BYOD has skyrocketed in recent years; with a recent survey by Bitglass claiming that over 82% of companies employ some form of BYOD [36].

Other than obvious financial savings from not having to purchase an additional mobile device per employee, BYOD improves employee productivity, work-life balance, and flexibility [19]. Despite these benefits, many organizations are hesitant to implement BYOD, mainly due to security concerns regarding sensitive company information and privacy concerns from their employees. [24]

Mobile device security, data breach security, mobile application security, and controlling employee use of mobile applications are all significant concerns when implementing BYOD [12]. Device and data security concerns mostly involve the risk of malware. Two examples of such malware, "Dresscode" and "xHelper", trick the user into downloading them by posing as legitimate applications. Once downloaded, the apps infiltrate whatever network a device connects to, including corporate networks. The malware attempts to download confidential files, possibly to be held as ransom or leverage for monetary gain. [38, 46]. Employee use and ownership of these devices also pose a large issue, considering their device may have access to sensitive company information.

In order to safely implement BYOD policy, network administrators need tools for monitoring devices for malicious activities. One such tool that was recently developed is Appjudicator. The tool captures UI interactions on an Android phone and associates them with 2 seconds of internet traffic immediately following the interaction. Appjudicator includes a built-in SDN agent containing flow rules for different packet types. When a cache-miss occurs, the SDN agent consults an off-device, supposedly a corporate-owned, SDN controller for a decision on a given packet.

Currently, Appjudicator monitors traffic and UI data from all applications on the device without any redaction. This would pose privacy concerns to anyone who has Appjudicator installed on their personal device as part of their company's BYOD policy. Appjudicator also faces technical challenges, including the inability to decrypt HTTPS traffic, which is used by over 80% of pages loaded by Firefox [31].

This Major Qualifying Project aims to extend the capabilities of Appjudica-

tor with the "PrivacyMonitor" module. PrivacyMonitor will explore methods in which Appjudicator can achieve a delicate balance of security, user-side transparency, and privacy.

From a technical standpoint, the PrivacyMonitor module will generalize network data in the form of batches sent to a new external system. These batches will be created based on matching basic packet information and UI data to set policies. A network operator using this external system will be able to set these policies in the form of policy updates and request more information regarding concerning policy hits. This system of policy matching will also affect how packets are routed on the device by leveraging Appjudicator's OpenFlow SDN Agent. Privacy Monitor will also provide support for TLS inspection or TLSi, which is a method for decrypting HTTPS traffic.

2 Background

Our approach builds upon Appjudicator as it currently stands, leveraging the Android Virtual Private Network (VPN) Service as well as networking and network security concepts such as VPNs, Software Defined Networking (SDN), OpenFlow, Mobile Device Management(MDM), HTTPS, TLS, and cryptography. We now explore each of these topics.

2.1 Appjudicator

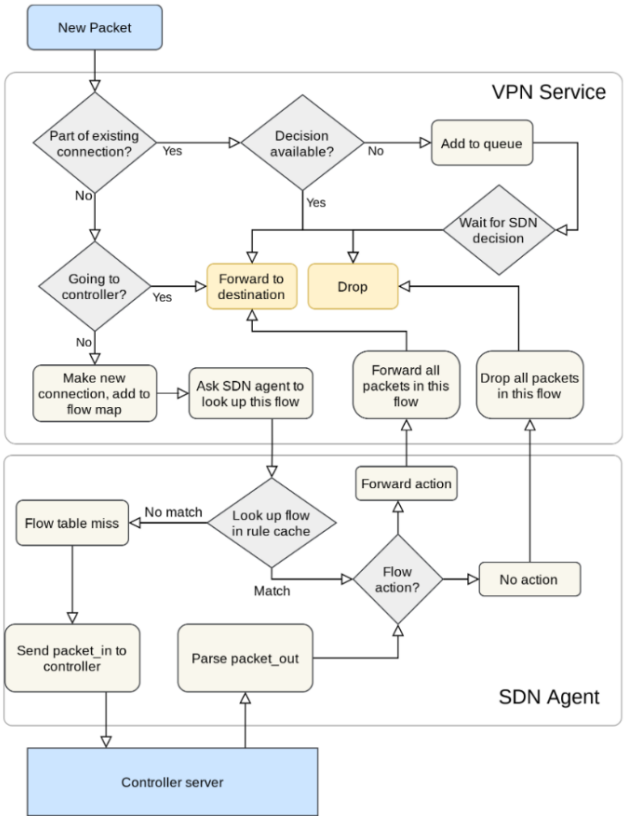

Figure 1: Basic Functionality of Appjudicator [23]

Appjudicator is an Android application designed to track network flows on the device. The app intercepts packets from the device by leveraging Android's VPN API, then decides whether or not to elevate the packet, recognizing it as the beginning of a new flow. The diagram above outlines the flow of packets through Appjudicator [23]

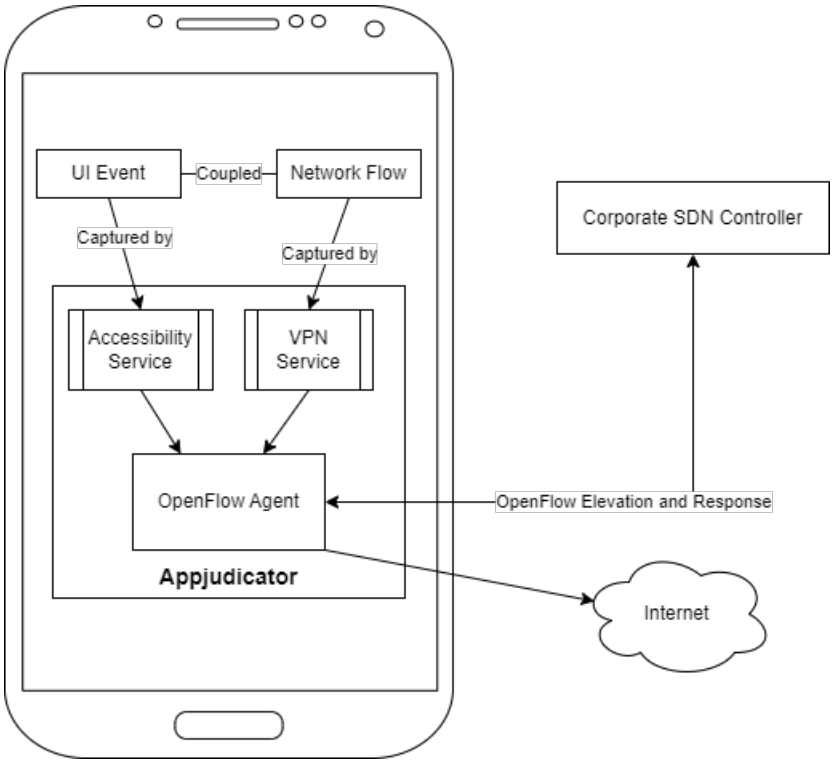

Figure 2: Appjudicator UI Interaction

Appjudicator also has the ability to associate user interaction with network traffic. Through the Android accessibility service API, each user interaction is logged and associated with any network traffic within a certain time frame. The figure above outlines the flow of information through Appjudicator, including the collection of UI context.

2.1.1 Android's VPN Service

Android's operating system architecture creates a virtual machine for each application to run in isolation. Although great for security, this presents challenges when capturing network packets on the device. To solve this issue, Appjudicator utilizes Android's Built-in VPN service. Appjudicator's VPN service creates a virtual network interface called a TUN (TUNnel) and configures addresses and routing rules, returning a file descriptor to Appjudicator. Each outgoing network packet is then forwarded to this new TUN interface, allowing Appjudicator to read them. Writing raw packets to the file descriptor forwards them to their

corresponding application based on IP address and port number. Android's VPN Service also allows the user to declare certain internet sockets as "protected" which allows them to circumvent the TUN interface, instead using the device's default network interface (commonly WiFi interface). This workflow is presented in Figure 3 [35].

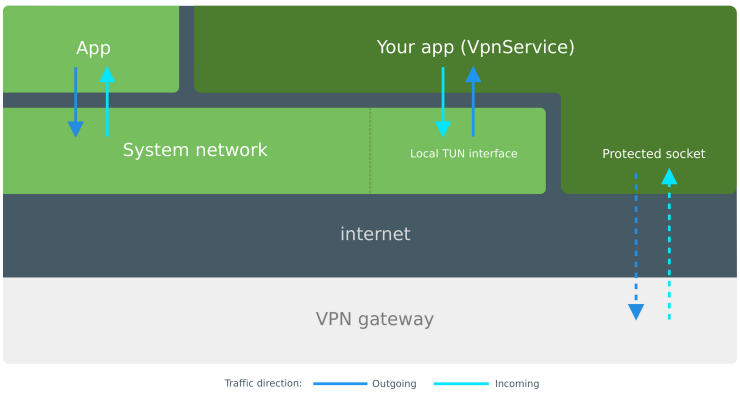

Figure 3: General Workflow of Android's VPN Service [35]

The intended use of the VPN service entails encrypting IP packets and sending them via a protected socket to a VPN server. However, Appjudicator simply uses the service to obtain raw IP packets for inspection.

2.2 Software Defined Networking (SDN)

SDNs allow for the dynamic and programmatic structuring of network configurations. This includes capabilities such as establishing policies, and assigning network settings, flows, and controls [13]. The virtualization of a network enables easy creation and testing of experimental infrastructures [20] Furthermore, the approach of SDNs gives way to better network adaptability, an important metric when evaluating its use in network security. Overall, the ability to customize and implement policies allows our system to privatize data and generate quick responses to potential network threats. Our system will leverage common SDN practices to ensure privacy and security.

2.3 Openflow

OpenFlow is a communications protocol that enables the establishment of centralized SDN architecture [7] The connection between a network controller and a device is handled by OpenFlow. Network traffic is controlled by a set of instructions called flow rules that determine how internet packets are routed. Allowing and denying network flows is done through matching criteria. The criteria for matching involves network packet information ranging from IP addresses to MAC and port numbers. Flow rules and their associated actions are handled by an OpenFlow controller.

In order to establish these rules the OpenFlow controller, typically a script, programs the flow rules which are then enforced and implemented by the OpenFlow Switch (OVS). Switches and routers act as the forwarding plane to the control plane logic controlled by OpenFlow's cache of rules and controllers [32] These switches are able to install, match and modify the rules it receives from the controller. Packets that miss on rule cache checks are elevated to the OpenFlow controller, which allows analysis to be done on packets and creates rules for them. OpenFlow allows packet data, network transmission routes, and protocols to be controlled independently, which will be leveraged in our approach.

2.4 Corporate Mobile Protections

In modern times mobile devices are ubiquitous in the workplace. Many models have been proposed for incorporating personal devices, like phones, into corporate systems. A survey by CyberlinkASP reveals that 95% of companies today allow bring-your-own-device (BYOD) in one form or another [19]. Although BYOD boosts productivity in the workplace, it poses new challenges from a security and privacy standpoint. Devices that connect to other networks introduce risks outside the control of the company. Specifically, BYOD devices can get infected by malware while not on the company network. The malware can later disrupt the company's network once the device connects to it. To detect this type of malware, device usage has to be monitored to different extents, and each of these comes with its own privacy concerns.

2.5 Mobile Device Management (MDM)

MDM is a system in which an organization (commonly a corporation) installs management software on personal devices or distributes company-owned devices

with management software to its employees. Administrators then have the authorization to set compliance rules on that device. If a device is not or becomes no longer compliant with an organization's rules, that device can be blocked from the network, or in more drastic situations, controlled directly by administrators. MDM tools are often too restrictive and invasive for a hybrid personal-professional device. Mobile Application Management (MAM) tools attempt to secure specific applications on an otherwise uncontrolled device, however, MAM cannot detect threats across mixed-use applications, such as search engines and messaging apps [27].

With the aforementioned concerns in mind, an improved BYOD system addresses the security and privacy concerns that employers and employees face. Furthermore, it eliminates the need for alternative systems such as Choose Your Own Device, corporate-owned/personally enabled devices, and corporate-owned business-owned devices. Having a gradient of acceptable network communications described by these security frameworks on a single device allows more flexibility for both IT departments and end users.

2.6 Cryptographic Hashes

A cryptographic hash is a function that takes a message of arbitrary length as input and produces a digest of a specific size called a hash. This digest consists of a seemingly random string of bytes. A cryptographic hash has three main properties:

1. For any specific output of the function it is computationally infeasible to find the input which produced it

2. Given any specific input to the function and the corresponding output, it is computationally infeasible to find another input for which the function produces the same output

3. It is computationally infeasible to find two different inputs to the function such that their hashes are equal.

The three properties mentioned above make hashes useful for verifying the integrity of data. If one communicating party produces a cryptographic hash for a message and sends it alongside the message, the receiving party can then verify that the message hasn't been modified by re-computing the cryptographic hash for the message and comparing it to the supplied hash.

2.7 Secret Key Cryptography

In computer science, cryptography refers to secure information and communication. Secret key cryptography is a type of cryptography where two or more parties share a key or secret which is only known to them in order to create secure communication.

Encryption is a cryptographic operation that encodes a message into a seemingly random sequence of bytes called a ciphertext in such a way that makes it infeasible for a third party to decode the original message. Decryption is the inverse operation of encryption. In symmetric key encryption schemes, two or more endpoints use a secret key and certain cryptographic operations in order to encrypt messages and decrypt ciphertexts. Theoretically speaking, a third party's only chance at decrypting ciphertexts is if they can somehow obtain the secret key.

2.8 Public Key Cryptography

Although secret key cryptography prevents third parties from deciphering messages, its security relies entirely on the secure exchange of the secret key, which is nearly impossible to guarantee. Communicating parties could exchange the secret in person, but as communicating parties are usually separated by vast distances, exchanging secrets in person becomes impractical.

Public key cryptography involves each communicating party having two keys that are linked, a public key and a private key. As its name suggests, the public key is publicly known to all parties including third parties. The private key is only known to the communicating party to which it belongs.

The private and public keys are linked in such a way that they are inverses to each other with respect to some mathematical operation. This operation has to be such that it is computationally infeasible to derive the private key from just knowing the public key. This operation is also different for different types of public key cryptography schemes.

Due to these special properties, a communicating party can use these operations to generate a ciphertext using the public key and their message in such a way that the ciphertext can only be decrypted by someone who possesses the corresponding private key. This is known as asymmetric key encryption.

Similarly, a communicating party can use its private key to encrypt a message. The receiving party will then be able to decrypt this ciphertext using the sending party's public key. This is useful because the receiving party gets a

guarantee about the authenticity of the message, as only a party possessing the correct private key could have produced the ciphertext. This process is referred to as a digital signature scheme, and it is used for authenticating messages similar to how a signature authenticates documents.

It is important to note public key operations are roughly 1000 times slower than secret key operations. Therefore, in practice, public key cryptography is used to exchange a secret key which is then used for communication via secret key cryptography.

2.8.1 Secure Socket Layer (SSL) Certificates

Lack of authentication presents a challenge for exchanging public keys over a network. For example, suppose a person Bob is trying to connect to a server. Bob wants secure communication, so he asks the server for a public key. Now, suppose another person Alice is snooping on Bob's packets. When she finds Bob's request for the server's public key, she simply impersonates the server by sending her own public key to Bob. In this public-key exchange, Bob would have no way of verifying that the key he received was from Alice and not the server. He would therefore continue the communication as normal, and possibly send Alice sensitive information.

SSL Certificates provide the ability to verify that public keys belong to a specific endpoint. For this to work, there has to be an authoritative figure whom everyone can trust. A Certificate Authority or CA provides SSL Certificates.

To request an SSL Certificate, an endpoint first has its public and private keys generated. It then sends a request to the CA which contains the public key, its domain name, and other information necessary for verifying its identity. If verification is successful, the CA provides the endpoint with an SSL certificate. This certificate contains a public key and domain name which are all signed by the CA's private key, after which the CA's info is appended. This certificate is only valid for a limited period of time, after which the endpoint must request a new certificate.

An SSL certificate can be issued on one's own behalf. These certificates are called self-signed certificates. Certain types of CA's have self-signed certificates, which are referred to as root CA's.

2.8.2 Certificate Chain of Trust

Currently, the trust model used for most public key infrastructure is the hierarchical model. In this model, there are three types of entities: root CAs, intermediate CAs, and end entities.

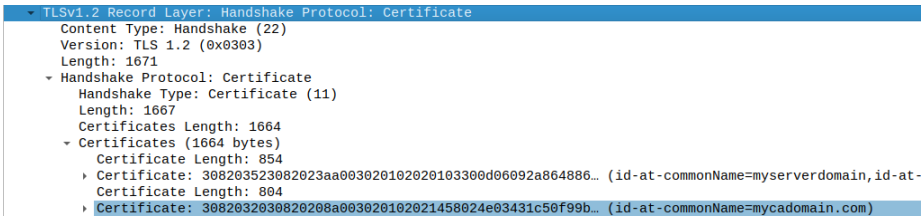

Figure 4: Hierarchical Certificate Chain

The end entities are endpoints that request certificates for their public keys. Their certificates are signed by the intermediate CAs. Intermediate CAs are CAs that provide certificates for end entities or other intermediate CAs. Their certificates are signed by either root CAs or Intermediate CAs which are higher up in the hierarchy. The root CAs are the most trusted authorities, as they are at the top of the hierarchy and therefore the root of trust. The certificates of root CAs are self-signed. A client commonly has a list of all trusted root certificate authorities and their public keys pre-installed on their operating system.

When a client receives an SSL certificate from a server, the client goes through this list of trusted CAs and matches it with the root CA stated on the certificate. The client then uses the CA's public key to verify that the certificate used to sign the next certificate lower in the chain is valid. After validation, the client then extracts that certificate's public key and repeats the process all the way down the chain until it reaches the endpoint's certificate. It then finally checks whether the domain name contained in the certificate matches the domain name of the endpoint the client is communicating with. If so, the client can then be certain that the public key inside the certificate does indeed belong to the endpoint the client is communicating with.

2.8.3 Certificate Revocation

When a private key of an endpoint is leaked, such as with the Heartbleed attack [37], an attacker then may use the certificate associated with the leaked private key to impersonate the endpoint to which the certificate belongs.

To mitigate this, certificate authorities that issued the certificate for the given endpoint can revoke the certificate before its expiration date. These revoked certificates are kept in a Certificate Revocation List (CRL) and are digitally signed by the CA [45]. Liu *et. al.* found that most desktop web browsers and all mobile platform browsers failed to check CRLs for revoked certificates [11] which is a major security risk. We revisit Certificate Revocation in the Limitations and Future Work of TLSi.

2.9 Transport Layer Security and HyperText Transfer Protocol Secure (HTTPS)

Hypertext Transfer Protocol (HTTP) is an application-layer protocol used by most web pages for transferring web page content to a user's computer. The HTTP protocol can transfer text, images, video, audio, etc. However, HTTP is unencrypted, making it a security risk.

If a malicious actor snoops on packets on a network switch, they are able to determine all of the contents of a web page some client is viewing. The malicious actor could also steal any sensitive information sent by the client such as credit card numbers, modify a client's message to the server, or masquerade as the web page in order to trick the client into sharing more sensitive information.

To mitigate security risks, the Secured Sockets Layer protocol or SSL was introduced in 1994 and later standardized as the TLS protocol. The TLS protocol uses two types of encryption: secret-key encryption for encrypting application data packets, and public-key encryption for exchanging encryption keys for secret-key encryption. HTTPS is HTTP which uses TLS or SSL for encrypted communication.

Before any application data can be securely exchanged, the TLS protocol first performs a TLS handshake to establish secret-key encryption parameters. After the handshake, these parameters are used to encrypt and decrypt application data.

2.9.1 TLS Cipher Suites

Cipher suites identify the different key exchange, signature, and symmetric encryption algorithms and hash functions that will be used during the TLS Handshake and further communication. Cipher suites are formatted as in Figure 5.

Figure 5: An example of a TLS Cipher Suite Format

In green is the key exchange algorithm. This specifies the public key exchange scheme that the client and server use to exchange the pre-master secret from which the secret-key encryption parameters can be derived. In yellow is the authentication algorithm. This specifies how the client is going to verify the identity of the server (and in some cases how the server is going to verify the identity of the client). In purple are the secret-key encryption algorithm, the length (or strength) of its key, and its mode of operation. This is the algorithm the client and server use to encrypt application data after the handshake is complete. Finally, in blue is the hash algorithm that will be used to ensure the integrity of the exchanged encrypted messages.

For this project, we used the "ECDHE-RSA-AES128-GCM-SHA256" cipher suite. Our reasoning for this choice is explained in the Methodology section.

2.9.2 TLS versions

TLS was first defined in 1999 in RFC 2246 [1]. Since then it has seen many revisions and changes. Notable changes include the updated TLS 1.1 version defined in RFC 4346 [4] in 2006, and then later TLS 1.2 in 2008 (RFC 5246 [8]) and TLS 1.3 in 2018 (RFC 8446 [18]). TLS 1.0 and TLS 1.1 are deprecated due to the discovery of various security vulnerabilities within the versions. TLS 1.3 is the current industry standard with a reported 63% of servers preferring TLS 1.3 to other protocols as of August 2021 [R22]. The key differences between TLS 1.3 and TLS 1.2 are that TLS 1.3 offers a shorter and simpler handshake; it completely removes support for cipher suites which are now considered insecure; it separates the key agreement and authentication algorithms from the cipher suites; and it removes support for some obsolete features. TLS versions have also been built with backward compatibility in mind, and TLS 1.2 is still considered reasonably safe in 2023 and most web servers (and clients) offer backward compatibility from TLS 1.3 to TLS 1.2. Therefore, this project primarily focuses on implementing the server-side code for TLS 1.2.

2.9.3 TLS Record Layers

Each TLS message sent consists of one or more record layers. A record layer consists of a header and payload. This header specifies the TLS version, the type of TLS message contained in the payload, and the length of the payload.

The types of payload are one of four: handshake messages, change cipher spec messages, application data messages, alert messages

We describe each briefly here:

1. Handshake messages are used at the beginning of a TLS connection. They contain information used to authenticate both endpoints and information that the two endpoints use to securely negotiate a symmetric encryption key. They also include information for ensuring the integrity of the exchanged information.

2. Change Cipher Spec messages are used by an endpoint to signal to the other that it is switching to encrypted communication which uses the newly negotiated symmetric key. All TLS record layers following this message will be encrypted.

3. Application Data messages carry application data encrypted using the newly negotiated symmetric key.

4. Alert messages are used to send different signals to the other endpoint. In most cases, they communicate errors.

Note: Each TLS Record layer is supposed to be compressed via a lossless compression algorithm. In practice, only the default, null-compression algorithm was used. In TLS 1.3 the compression of the Record Layer has been deprecated.

2.10 TLS Handshake Message Subtypes

The Handshake messages for TLS 1.2 also have 8 sub-types. We will go over them briefly here as well:

1. A Client Hello Message is the first message in the handshake and it initiates the handshake process. In this message, the client specifies its highest supported TLS versions, supported TLS cipher suites and compression methods, as well as other features and extensions. Extensions can have different effects, some specify supported point formats for elliptic curves, others specify how the master secret is computed, etc. The client hello

message also includes a 32-byte random number used in later computations for preventing certain types of attacks. The session ID number is sent by the client to resume a prior TLS session using the same keys. A screenshot of how Wireshark parses the Client Hello Message is shown in Figure 6:

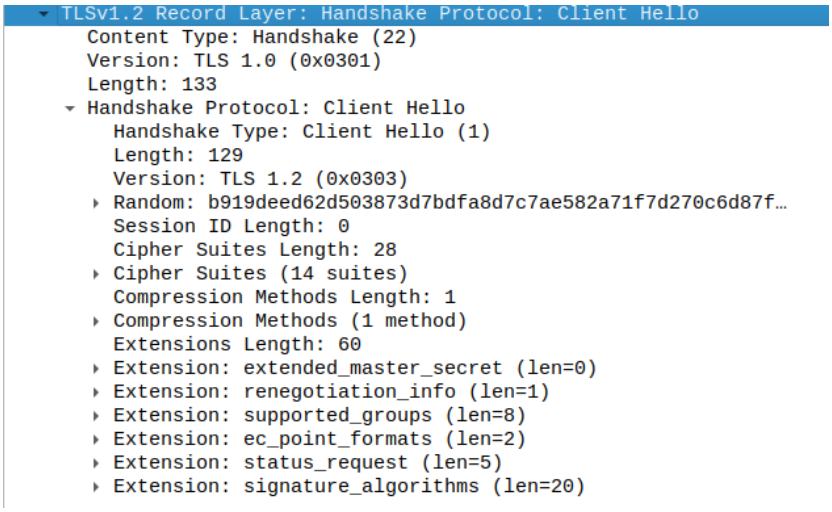

Figure 6: Client Hello Message parsed by Wireshark

2. A Server Hello Message is the first TLS record layer the server responds with. Within the message, the server specifies a TLS version, cipher suite, and compression method that both endpoints will use. The server hello message also includes a 32-byte session ID and its own 32-byte random number. If the client did not specify a session ID, or if the server does not recognize the session ID provided by the client, it generates a new one. A screenshot of how Wireshark parses the Client Hello Message is shown in Figure 7:

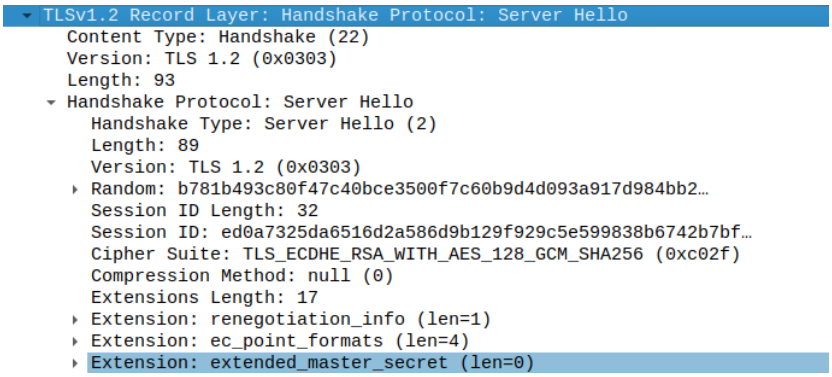

Figure 7: Server Hello Message parsed by Wireshark

3. A Certificate Message is a message containing an SSL certificate for authentication. This message is usually sent only by the server unless the server requests that the client authenticates itself. A screenshot of how Wireshark parses the Certificate Message is shown in Figure 8. For development purposes it is important to note that these certificates are in Distinguished Encoding Rules (DER) format:

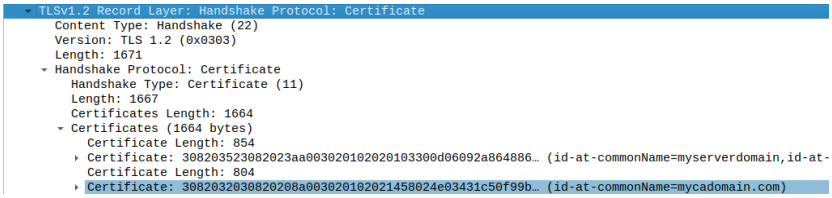

Figure 8: Certificate Message parsed by Wireshark

4. A Server Key Exchange Message is sent by the server and includes the server's public key information and other parameters for the key exchange algorithm of the chosen cipher suite. This public key information is signed via a digital signature scheme which is also specified in the message. A screenshot of how Wireshark parses the Server Key Exchange Message is shown in Figure 9:

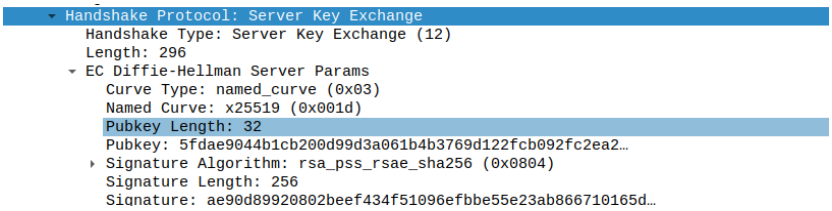

Figure 9: Server Key Exchange Message parsed by Wireshark

5. A Server Hello Done Message follows the server key exchange message and signals that the server is finished with its hello message. A screenshot of how Wireshark parses the Server Hello Done Message is shown in Figure 10:

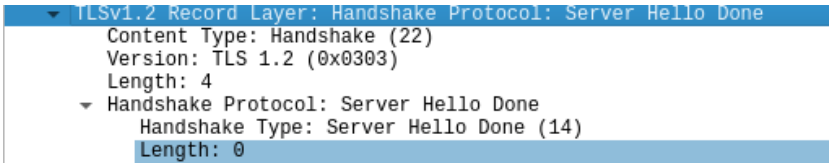

Figure 10: Server Hello Done Message parsed by Wireshark

6. A Client Key Exchange Message is similar in function to the server key exchange, but is instead used by the client to send its public key information to the server. It usually does not include a signature (unless the server is also authenticating the client). A screenshot of how Wireshark parses the Client Key Exchange Message is shown in Figure 11:

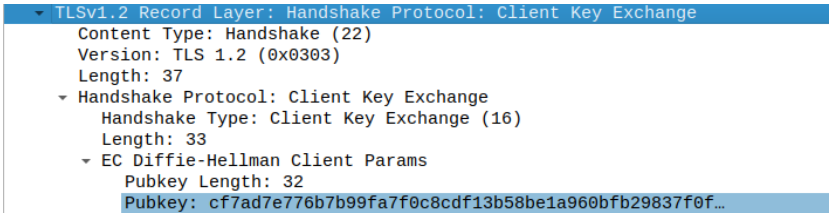

Figure 11: Client Key Exchange Message parsed by Wireshark

7,8 Client Finished and Server Finished messages contain a message authentication code called "verify_data" which is encrypted via the symmetric encryption scheme. A screenshot of how Wireshark parses the encrypted Client Finished is shown in Figure 12. By instructing OpenSSL's s_server

functionality to output TLS keys, it is possible to decrypt the Finished messages in Wireshark, as shown in Figure 13:

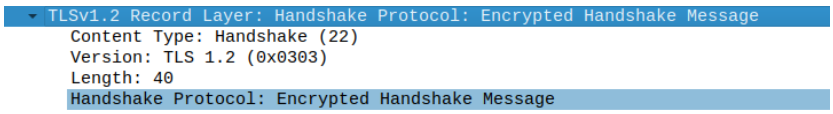

Figure 12: Encrypted Finished Message parsed by Wireshark

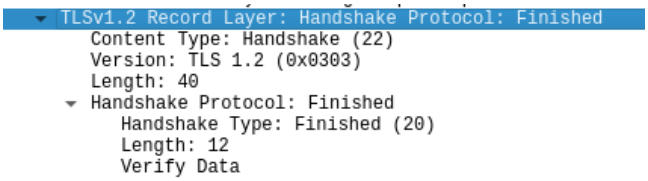

Figure 13: Decrypted Finished Message parsed by Wireshark

3 Related Work

In this section adjacent technologies and prior work done in the privacy field is discussed. These concepts serve to better inform our results and findings.

3.1 Competing Technologies

There are many existing software with the goal of defending a network from mobile devices. Popular industry MDM software includes Microsoft's Intune [27], which is a cloud-based endpoint management solution that aims to secure corporate networks through conditional access policies and remote control. However, Intune has been the subject of many criticisms, mostly for being invasive or overbearing [17].

Notable academic software includes AppIntent [10], which explores whether data transmissions of an application are leaking private information. This is achieved by attempting to discern if data transmissions are user-intended through static analysis and dynamic execution event handling. Another example of academic software is AppContext [14], which examines malicious application mimicry and strategies to evade detection through static analysis. While both of these technologies make significant contributions to the field, PrivacyMonitor

is unique in its user-defined expectation of privacy and does not require to be rooted in the device, as PrivacyMonitor does not perform static analysis.

3.2 Private Information through Contextual Integrity

The project goals include protecting a user's private information from being monitored in privacy breaking ways. We define private information based on Nisselbaum's notion of contextual integrity [3], which is a conceptual framework for understanding privacy expectations and their different implications. The framework's goal is to " formalize concepts from contextual integrity so that privacy guidelines, policies, and expectations can be stated precisely, compared, and enforced by an information processing system." [3].

This framework allows us to evaluate the flow of information between two agents in any scenario. Consequently, sensitive-attributes in privacy metric calculations can be determined with this framework. It currently informs numerous privacy legislation in effect today, including the Health Insurance Portability and Accountability Act (HIPAA), the Children's Online Privacy Protection Act (COPPA), and the Gramm–Leach–Bliley Act (GLBA) [3].

3.3 K-Anonymity

The metric of k-Anonymity measures the re-identification risk of records, assuming a structured data set that is specific to individuals. With k-anonymity, there exists a guarantee that each record is linked to at least k-1 other individuals in the data set who share attributes [6]. Through this practice, any analysis of the dataset that reveals potentially privacy-invasive information is associated with at least k-1 other records. This obfuscates private information while keeping the data's content relevant for additional security goals.

3.4 L-diversity

L-diversity is another data privatization technique that restricts access to sensitive information by introducing granularity to the attributes of a dataset [5]. In this way, sensitive attributes by themselves are unable to disclose the identities of individuals in the dataset. In order to make the attributes sufficiently diverse, the data is partitioned into equivalence classes. These classes will then have at minimum L distinct values for each sensitive attribute. So long as the classes

are sufficiently descriptive, yet still generalized, the dataset will be both useful for analysis and privacy preservation.

3.5 Differential Privacy

Another standardized privacy preservation technique that sees use in the information security field is differential privacy. Similarly to k-anonymity and l-diversity, differential privacy ensures that datasets are able to store potentially private information without the risk of revealing identifying information. In order to test and implement this, artificial data is generated and included in a synthetic dataset alongside the original dataset [26]. The random noise added to the dataset makes queries against the database for an individual's information unidentifiable. At the same time, the size of the synthetic information is small enough to keep the analysis on the dataset relevant.

A dataset is said to be differentially private if any individual record coming from the synthetic or organic datasets is indistinguishable from one another. Consequently, even given an individual with wholly unique attributes in the dataset, their data will be guaranteed to retain privacy through differential privacy [22]. With a differentially private dataset individual record information is protected against leaks while retaining the data's utility.

4 Methodology

In this section we outline the details of how Privacy Monitor and TLSi have been implemented. We start first with Privacy Monitor's workflow and design, then explore the TLS Inspector's implementation.

4.1 General Workflow of Privacy Monitor

The general workflow of Privacy Monitor is depicted in Figure 14:

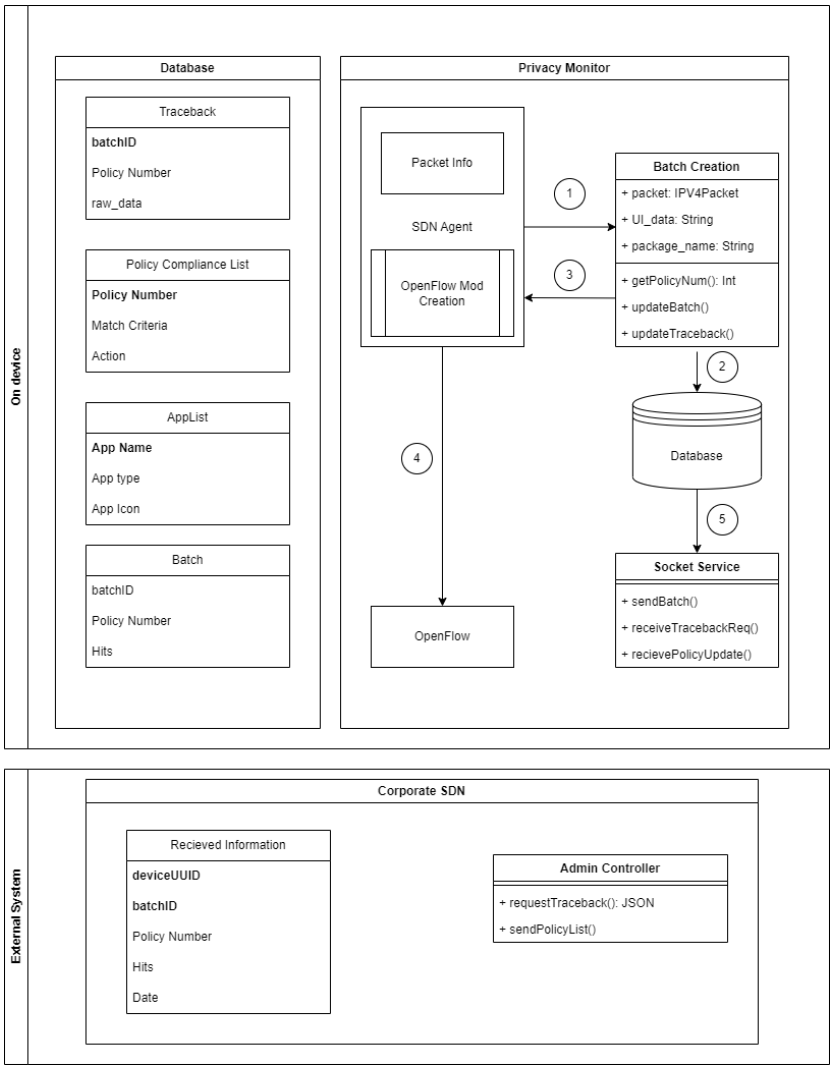

Figure 14: High-Level Overview of System Architecture

Before the beginning of our workflow, the user is expected to classify each of their installed applications as private-use, mixed-use, or corporate-use. The impact of these classifications will be discussed in greater detail later.

The operation of the system starts with Appjudicator deciding to elevate a packet from a new network flow. This packet arrives at the SDN Agent, where our workflow begins.

1. Appjudicator's existing framework allows us to obtain relevant packet and

UI information and forward it to the Batch Creation module.

2. The Batch Creation module then takes the information and matches it to a policy using the Policy Compliance List. The Batch and the Traceback tables are updated accordingly.

3. The Batch Creation module returns the policy number associated with the information given to Appjudicator's SDN Agent.

4. The SDN Agent checks if that policy number is contained in the Policy Compliance List table, and reads its match criteria and action fields in order to determine which values to use when constructing an OpenFlow packet. Appjudicator's workflow continues as normal, updating the device's Flow Table.

5. The Socket Service is responsible for communication between the android device and our external system. It periodically reads the Batch information from the database and prepares to send it to our External System.

The External System stores the batches along with the device UUID and date received. It also has the ability to request more information regarding a certain batch with a Traceback request and send information to update the phone's Policy Compliance List. This is handled by the SocketService.

4.2 Graphical User Interface (GUI)

Privacy Monitor allows the user to classify each application with internet access on their device through a new "Monitor" menu in Appjudicator. The user is able to easily scroll through their apps and modify/check their classifications. Since these classifications affect how data is retained and handled, allowing the user to set them for each app provides a method for defining their expectation of privacy on a per-app basis.

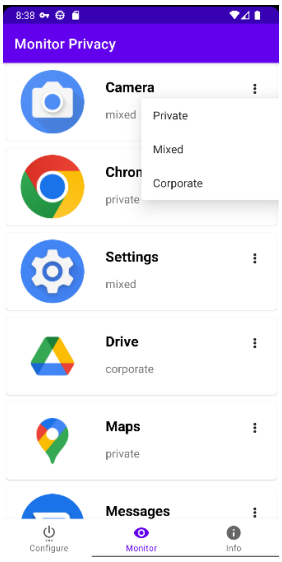

Figure 15: Screenshot of Privacy Monitor's GUI

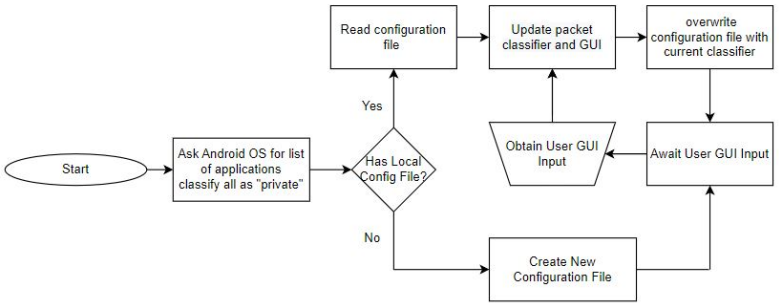

Figure 16: Flowchart of GUI Operations

Privacy Monitor first queries all applications on an android device and only displays those with networking capabilities. This is done using the "GET_TASKS" permission and Android's built-in "PacketManager" class [34].

Once the list of applications is obtained, all applications are initially classified as "private". Privacy Monitor then checks its database for existing configurations. This file contains a list of applications and their classifications. If there is no saved classification for a given application, its classification stays private.

Privacy Monitor then awaits user GUI inputs for changing classification set-

tings for individual applications. Once a GUI input is detected, the database and GUI are updated.

All of the applications are displayed in a scrollable list, where on the left the application icon and name are displayed, and on the right one of three keywords: "personal", "mixed" or "corporate" referring to the classification of the given app.

4.3 Batch System

Instead of sending full packet information and appended UI data, PrivacyMonitor generalizes the information into batch updates for the network administrator to monitor. The full information is then logged locally. Batch information matches packet information to pre-defined policy numbers. Batches are sent every 30 seconds to the external system, which stores them.

Network administrators will be able to view batches and react to concerns through trace-back requests and issuing policy compliance lists. These mechanisms will restrict the user's access to certain resources and allow network operators to investigate alarming network activity.

4.3.1 Batch Creation and Impact of App Classifications

Appjudicator associates accessibility information with an elevated network packet and appends the accessibility information to the end of its payload before sending it to an off-device controller. Privacy Monitor uses this information as well as its app classifications to create batches.

PrivacyMonitor sends batches in the form of .json files to the external system every 30 seconds. These batches will have a field for the ID of the batch, and fields for each policy containing how many matches, or hits, were found. These matching rules are designed to be customized and can be changed based on what a network operator is looking for.

The policies can be defined based on the following extracted information:

1. IP Source and/or Destination Address
2. Source Port and/or Destination Port
3. Presence of a UI Event or Specific UI event
4. Classification of the Application from which traffic was generated

Policies must include at least one of these matching criteria and may use as many of these criteria as desired. As an example, Privacy Monitor allows a policy to be defined for when a mixed-use application attempts to access a specific IP, with the network traffic being associated with the specific "TYPE_VIEW_CLICKED" UI event.

App classifications will impact data retention at the point of collection in the following manner:

1. Private applications will never store specific UI data nor the name of the application the traffic originated from
2. Mixed applications will always store specific UI data but will not store the name of the application the traffic originated from
3. Corporate applications will always store specific UI data and the name of the application the traffic originated from

4.3.2 Policy Compliance List

Network operators are able to ultimately affect how the device's OpenFlow table is modified through the issuance of a Policy Compliance List. An entry in this list specifies a policy number, match criteria, and a desired action when those match criteria are met.

1. Policy Number: Number assigned to the new Policy
2. Match Criteria: Consistent with aforementioned Batch Creation matching criteria
3. Action: Variable to inform how OpenFlow packet is crafted

The match criteria given by the compliance item are consistent with the matching criteria in the batch creation. The action field given by the compliance affects how the resulting OpenFlow packet is crafted, therefore affecting the device's FlowTable and how the packet is routed as Appjudicator's workflow continues.

4.3.3 Traceback Requests

When batch hits are created, the full packet and associated UI data are written to a Traceback table in the database.

Network Operators are able to request more information associated with specific batches they receive using their batch IDs and the UUID of the device from which the batch was received, and the policy number of the hit being investigated. This feature is designed to be used only when there is a concerning policy hit to the network operator.

4.4 TLS Inspector

Next we look at how TLS Inspector was implemented. First describe the libraries used for its creation, and then we explain its workflow and main obstacles in creating a TLSi system in Android.

4.4.1 Tools and Libraries Used

We used several libraries and tools during the development of the TLS Inspector. We present each one briefly below:

BouncyCastle [30] is a cryptographic library developed for Java and C# which provides many functionalities used for developing cryptographic solutions. We used it heavily during the development of the TLS Inspector component along with Android's built-in Java Security library.

OpenSSL [29] is a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication implemented in C. For our purposes, we had to manually build the OpenSSL library with the "TRACE" option enabled. This allowed us to compare and test our own implementation of TLS against a commercial-grade implementation, which sped up development.

Wireshark [33] is a networking tool used to capture packets on Desktop computers. It also provides parsing capabilities for TLS packets. The TLS Inspector was primarily implemented in Java.

4.4.2 TLS Inspector Overview and Main Challenges

Given that most modern-day traffic uses the TLS protocol for encrypted communication, it is imperative that Appjudicator implements functionality for decrypting such traffic. Steps in The TLS protocol can be inspected under special circumstances by an on-path agent which then enables the decryption of the TLS data. This process is known as TLS Inspection or TLSi.

A high-level overview of performing TLSi is presented in Figure 17.

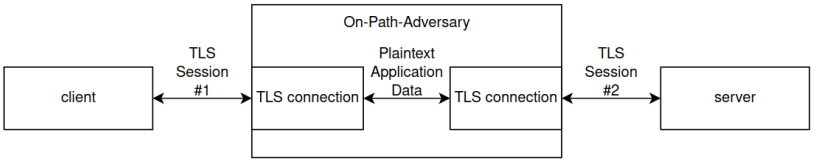

Figure 17: High Level Overview of TLS Inspection

During a connection between a client and server, if an on-path-agent manages to intercept a TLS handshake, the agent is able to create two TLS connections: one between the agent and the client, where the agent pretends to be the legitimate server; and one between the agent and the server where the agent pretends to be the client. The agent can then obtain application data from the legitimate client and server and simply pass them between the two newly created TLS sessions. This allows us to decrypt the TLS traffic without alerting the client or server of any wrongdoing.

The connection between the on-path-agent and the legitimate server, where the server does not validate the identity of the client (which is most TLS connections), is trivial to implement via existing Java libraries available on Android. These libraries support the creation of SSL Sockets, i.e. sockets that allow for encrypted communication via a specified protocol (in our case TLS). Due to the intricacies of how Appjudicator establishes connections with remote servers, this connection with the remote server needs to be established via a SocketChannel [42] rather than a traditional Socket [41]. The difference between the two is that a Socket [41] uses blocking network calls, whereas a SocketChannel [42] uses non=blocking calls. The challenge here is that standard Java libraries only provide support for an SSLSocket [44] and not an SSLSocketChannel. Luckily, we found an SSLSocketChannel implementation made publicly available on GitHub [28] and slightly modified its NioSslClient for our purposes. For this project in particular, creating the connection between the on-path-agent and the client app was much more difficult due to two main obstacles:

Obstacle 1 : During the TLS handshake, the client always verifies the identity of the server via a digital signature and a certificate containing the server's public key provided by the server. As described in the Digital Signatures Section in the Background, the client holds the self-signed certificate of the root CA at the end of the certificate chain of the server's certificate and uses it to verify that the public key provided by the server does indeed

belong to a trusted source. We must therefore provide a legitimate-looking certificate provided by one of the root CA's installed on the android phone to the client application and use the associated private key to create digital signatures.

Obstacle 2 : In our application, the Android VPN Service sends and receives raw IP packets via the TUN interface to and from client applications on the device. This is the core problem the TLS Inspector attempts to tackle. The reason why this problem is non-trivial is that it prevents us from using SSL Sockets from the Android library, which already implements the TLS protocol for all TLS versions and cipher suites. The SSL Socket object completely abstracts away the TLS Handshake messages and only presents the developer with API for sending application data via a secure channel. In other words, it does not provide a way for the developer to supply it with a TLS Handshake message, for which it would create an appropriate TLS Handshake response. In retrospect, there may exist a workaround that we missed that will be discussed further in the limitations section of this document. This alternative may have been much simpler from an implementation standpoint but ended up not being feasible.

Obstacle #1, although non-trivial, is still fairly simple to implement. For our implementation, due to time constraints, we wanted to make things as simple as possible, so we opted for directly installing the self-signed root certificate inside the Emulated Device's keystore via copy and paste, and then hard-coded its private key into the TLS Inspector component.

A more elegant solution would have been to, upon first launch, have TLS Inspector generate a self-signed root certificate and private key and then prompt the user to install it. However, in newer versions of the Android operating system, it may not be possible to use such a newly installed root certificate for our purposes.

After installing the root certificate in either fashion, when a client application initiates the handshake, the TLS Inspector acts as a root CA which then generates a new certificate signed via this newly generated private key for the server the client app is trying to connect to. It sends this "fake" certificate back to the client in its first TLS Handshake message, posing as the destination server. Because the "fake" certificate is signed by a root CA the Emulated Android Device now trusts, the client will accept it, and will proceed with the TLS handshake.

Obstacle #2 becomes a much tougher challenge at the time of development, our team could not find a freely available Java (or Kotlin) library that takes in raw TLS handshake packets and produce an appropriate output. Therefore, we had to develop our own code which implements the TLS handshaking process and encrypted communication on the server side. The next obstacle in this process is that there also seemed to be no freely available Java libraries for parsing TLS packet information. The best that we could find was a library called "Pcap4J" [39] which went only as deep as the TCP protocol (and TLS is carried over TCP). Therefore for this project, we had to implement our own custom TLS packet parser and we modeled it loosely according to how Pcap4J did its packet parsing. Then, using the parser to extract the necessary information, we implemented code that performed the TLS Handshaking process on the server side and also provided correct raw TLS Handshake packets which could be sent back to the client applications.

For our implementation, we focused on implementing TLS 1.2 for reasons mentioned prior. We also decided to go with the ECDHE-RSA-AES-128-GCM-SHA256 cipher suite because it is a suite whose key exchange algorithm, authentication method, data encryption, and hash function are not deprecated and is all also supported in TLS 1.3. This should make future endeavors of expanding the project into supporting the TLS 1.3 handshake process smoother.

4.4.3 TLS Packet Parsing

For the purposes of implementing packet parsing we used RFC5246 [8] which first defined TLS 1.2. We then used OpenSSL's s_client and s_server utilities which used TLS 1.2 and our desired cipher suite and connected them to each other which gave us an example of proper TLS Handshake packet exchange captured in Wireshark. Based on this information and the information contained in RFC5246 we implemented our TLS packet parser.

4.5 TLS 1.2 Handshake Workflow & Sequence Of Messages

The messages and TLS record layers exchanged during a TLS 1.2 handshake are shown in Figure 18.

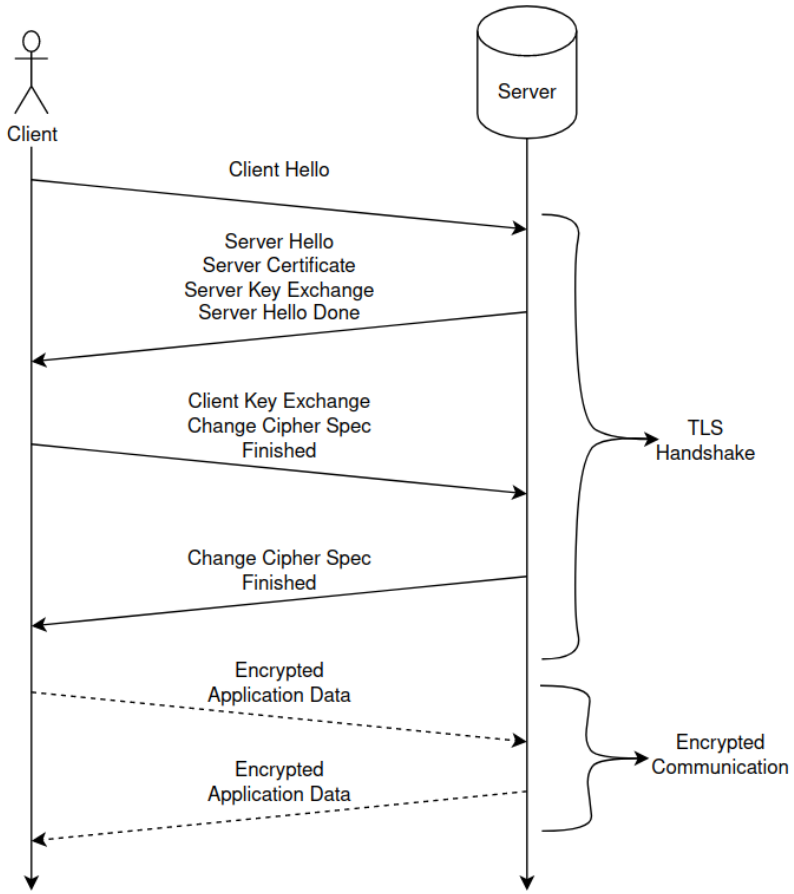

Figure 18: Messages and TLS Record Layers Exchanged in Handshake

In the next subsections, we will examine in finer detail these handshake messages and the record layers they contain. More specifically we will look at the TLS 1.2 Handshake for the ECDHE-RSA-AES-GCM-128-SHA-256 cipher suite, using the RSA-PSS-RSAE-SHA256 [34] signature algorithm, as these are the settings used for our project. We will also mention a few important implementation details which we found to have been left ambiguous in the RFC files.

4.5.1 Client's first Message

The first message the client exchanges is very simple as it contains only one TLS record layer, namely the Client Hello Handshake Message as discussed prior.

4.5.2 Server's First Message

The first message the server exchanges contains 4 record layers.

The first is the Server Hello Record Layer where the server specifies, among other things mentioned in the previous subsection, that it will use the ECDHE-RSA-AES-GCM-128-SHA-256 cipher suite. We noticed that the server usually needs to also include the "renegotiation info" extension which is used when re-handshaking. The RFC specifying the extension (RFC 5746 [9]) leaves it free for the client to decide whether they want to continue handshaking with a server that does not support the extension. We found that most industry standard TLS clients end the connection if this extension is not supported.

Since the cipher suite's authentication algorithm is RSA, the second record layer is the Certificate Handshake Message, where the server includes the certificate for its RSA public key which is authorized for digital signing. The server uses the private key corresponding to the public key in the certificate when creating the next record layer.

The third record layer is the server key exchange. For ECDHE the server must specify which curve it is going to use. In our case, it was the named curve "X25519". Next, the server generates a public-private key pair for the elliptic curve it chose. It then provides its public key (and it does not share its private key), a signature algorithm, and a signature inside the record layer.

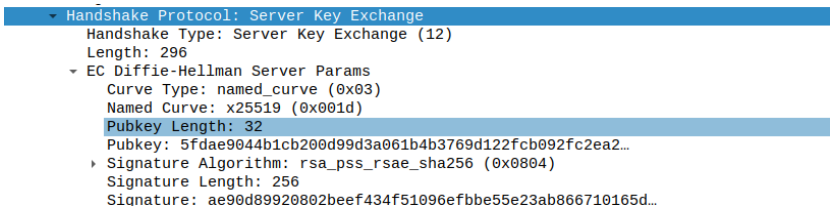

Figure 19: Server Key Exchange Parsed by Wireshark

The signature algorithm we have used is "RSA-PSS-RSAE-SHA256". There is no particular reason for this choice apart from the fact that while using industry-standard TLS implementations we noticed that android client applications support this signature method and commercial-grade TLS server imple-

mentations seem to prefer it. A screenshot of how Wireshark parses the Server Key Exchange is displayed in the figure above.

To construct the signature, the two 32-byte random numbers contained in the client and server hello messages are concatenated with the EC Diffie-Hellman Server Params (The Curve Type, Named Curve, Pubkey Length, and Pubkey bytes). This concatenation is known as the input message. We can construct the signature using the Java Security library Signature API [40] by getting the signature instance "SHA256withRSA/PSS". PSS stands for "Pseudorandom Signature Scheme", a signature scheme that uses a randomly generated salt. PSS parameters are then configured to use "SHA-256" as the hash algorithm and MGF1 as the mask-generating function. The Signature object is then initialized with the server's private key. The input message is first encoded via MGF1 and then signed to produce the signature. The signature algorithm as well as the MGF1 and other details are all described in RFC 8017 [15].

The final record layer the server sends in its first handshake message is the server hello done, to signify the end of the server's first handshake message.

4.5.3 Client's Second Message

Upon receiving the first handshake message from the server the client does a few things. It first checks the Server Hello Message and verifies that the server selected a cipher suite and compression method it supports. It then verifies that the server also supports the critical TLS extensions. In our case, these are the "renegotiation info" and "ec_point_formats" extensions. Once verified, the client then knows which cipher suite to use. In our cipher suite, we have specified that we are using RSA as the authentication algorithm and ECDHE as the key exchange algorithm. Therefore the client verifies that the next record layer is an RSA certificate which can be used for creating digital signatures. It then verifies that certificate's chain of trust.

The client then inspects the Server Key Exchange record layer. From the message, the client extracts the elliptic curve and the server's public key. It then runs a signature verification process described in RFC 8017 to verify that the signature is genuine. Once the signature is verified the client generates its own ECDHE public and private key pair. It then performs some elliptic curve operations to derive the pre-master secret. For the case of ECDHE and curve X25519, this is a 32-byte secret.

The master secret is derived from the pre-master secret. TLS 1.2 specifies

that the master secret is exactly 48 bytes for all cipher suites. The master secret is derived from the pre-master secret via the pseudo-random function (PRF) as described in RFC 5246. The PRF takes a secret, a label, and a seed as its inputs, and produces an output of arbitrary size. We will not go into detail about the PRF as its implementation is explained fairly clearly. We will state however that neither BouncyCastle, Java Security nor any other implementation of this function is widely available. Therefore we were required to write our own implementation.

Once the Master Key is derived the client then runs a key expansion, which expands the Master Secret into a key block of appropriate size via the PRF. This key block is partitioned into symmetric keys and implicit initialization values for the symmetric cipher chosen. For AES128-GCM, a key block of size 72 bytes is required. The first 32 bytes are the client symmetric key, and the next 32 bytes are the server symmetric key. The last 8 bytes are split between the implicit initialization values (IV's) for the client and server respectively. For GCM the entire IV is 12 bytes consisting of 4 implicit bytes (which never change unless a re-negotiation occurs), and of 8 explicit bytes, which are randomly generated for every encrypted message.

Once all of this is performed, the client prepares the second and final TLS handshake message from its side which contains 3 record layers.

The first record layer is the Client Key Exchange Message where the client includes its public ECDHE key.

The second record layer is the Change Cipher Spec Message, which just signals to the server that the next message is encrypted using AES-GCM128 and the negotiated symmetric keys and IVs.

The final record layer is the Client Finished Message which will be encrypted via AES-GCM128. The unencrypted message is shown again in the figure below.

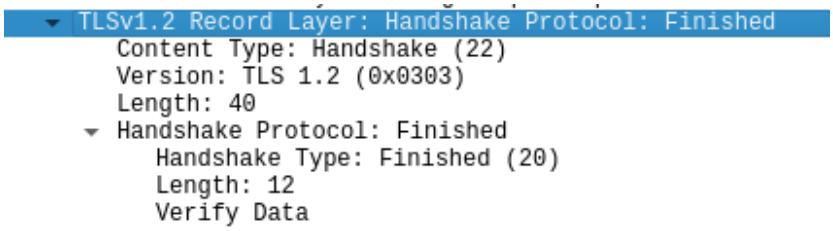

Figure 20: Unencrpyted Finished Message Parsed by Wireshark

The verify data field is 12 bytes of data serving as a message authentication

code. It is generated again via the PRF. We note that the seed as described in RFC 5246 is the SHA256 hash of the concatenation of all the handshake messages so far at the Handshake layer and below. For example, in Figure 11, this includes all the bytes from "Handshake Protocol: Client Key Exchange", but not the Record Layer headers (i.e. the Content-Type, Version, and Length fields). We also note that the Change Cipher Spec messages are not included as they are technically not a Handshake Message, but rather a separate category.

The client then generates 8 random bytes for the explicit initialization value. The verify data field and the header of the Finished message (i.e. the Handshake Type and Length fields) are all encrypted using AES-GCM128 and the client's symmetric key and the 12-byte IV (4 bytes implicit + 8 bytes explicit IV).

The GCM mode of operation also requires authentication data called known as the additional data as described in RFC 5246. This data is a concatenation of a sequence number and the record layer header which we are attempting to encrypt. There are two important things to note that were left ambiguous in RFC 5246. First, both the server and client keep a sequence number which counts how many record layers have been encrypted so far under the particular connection state. After the "Change Cipher Spec Message" this number is set to 0. Second, the Length field is the length of the unencrypted payload (in our case 16 bytes: 1 for the Handshake type, 3 for length, and 12 for verifying data).

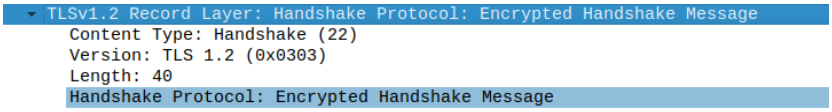

Figure 21: Encrypted Finished Message parsed by Wireshark

The output of this encryption is 32 bytes long, 16 for the encrypted data and another 16 for the MAC automatically produced by the GCM mode of operation. The client prepends 8 bytes of the explicit IV to these 32 bytes. This then becomes the Encrypted handshake message as shown in Figure 21.

4.5.4 Server's Second Message

Upon receiving the client's second message, the server inspects the Client Key Exchange record layer and extracts the client's public key. The server then repeats the same steps as the client to derive all of the symmetric keys and initialization values. The server verifies that there is a change cipher spec message.

The server then inspects the Finished message, which is now an Encrypted Handshake message. It extracts the first 8 bytes as the explicit client initialization value and decrypts the remaining 32 bytes using the client symmetric key and implicit IV. The server then verifies that the verify data field of the decrypted message matches the expected verify data it computes on its own end.

The server then prepares its final handshake message consisting of a change cipher spec and Server Finished message. The finished message is then also encrypted and sent to the client.

The client then decrypts the message and verifies that the verify data is what is expected. The TLS handshake is then finally complete.

When decrypting the message the endpoint must supply the additional data that the other end used as input to AES GCM to produce the authentication data. One part of this data is the length of the payload before encryption. It is always the length of the encrypted payload minus 22 bytes (8 for the explicit IV size and 16 for the MAC size).

5 Results and Findings

Here we present the experimental setups for testing Privacy Monitor and TLS Inpsector. We first go over the testing setup and results of testing the Privacy Monitor.

5.1 Measurement Taking Process

Below we outline the steps we will take to quantify and ensure our system meets the privacy goals presented in our paper:

1. Anonymity Set Size
    (a) Measure k-anonymity on IDs
        i. We want to ensure that given the IDs, policy number, and time this network flow took place that an individual phone cannot be singled out based on identifying attributes.
        ii. First sample data is generated through use of the system that we have built during development and an additional testing harness.

iii. We then make queries on the corporate SDN side to ensure that records are indistinguishable from K amount of other records. With our system giving UUID as sensitive information the combination of batch ID, policy number, and date should give us a very high K-anonymity number.

iv. After, the risk of re-identification in our database is calculated as the maximum probability of being re-identified which is 1/k [6]

v. Given our sample data we can measure our actual probability of re-identification with our ideal one to ensure our model does an accurate job of privatizing data.

(b) Measure L-diversity

i. We want to ensure that there are L-number of distinct values for the sensitive attribute group to make re-identification from a potential data breach more difficult. Specific types of accessibility info will be generalized into generic messages.

ii. To perform further analysis and calculations we use our previous sample dataset.

iii. Then a query will be made on corporate policies, date-time and accessibility service type from which l-diversity can be determined.

iv. From here a risk assessment of the data can be done with the phones UUID as our sensitive attribute along with our quasi-identifiers of, policy number, date and batch ID.

2. Data Retention Policies

(a) Implementation of a 2-week data-retention policy for both on-device traceback log database and corporate SDN batch database. This will ensure that data can only be accessed when it is relevant, minimizing the risk of leaks and preventing a large pool of user-specific information from being collected.

(b) Our data retention policy is based on company needs, and can be adjusted accordingly.

(c) This functionality test is vital for our aforementioned privacy metrics, such as k-anonymity and l-diversity since the retained logs will determine the values of these metrics.

3. Access Control and Data Minimization

   (a) Our batch system ensures the SDN receives a generalized form of information. A network operator also has the ability to investigate further into concerning network activity

   (b) Our batch system notifies on general information such as new network flows with or without user interaction

   In our future work section, we expound upon the concept of differential privacy and its applications in testing future datasets with PrivacyMonitor.

4. Batch Information Specifications and User-side Transparency

   (a) Our batch system classifies apps into 3 categories, private, mixed-use, and corporate. These classifications allow a user to decide how an application will be viewed and affected by the corporate SDN. This can include information such as general packet information, and specific ui-actions such as a button press.

5.2 Resource Usages

To gauge whether the system's goal of being lightweight is met we evaluate its resource usage. Thus, we evaluate our module based on energy expenditure, in the form of battery usage, along with CPU and memory usage. Android emulator enables testing for these metrics on Appjudicator with the companion PrivacyMonitor module.

Android Studio Profiler [21] allows the monitoring of the phone's resources on a per-app basis. Tests were run on a laptop computer with 6 cores and an allocated 11 GB of memory. The emulator simulated a Google Pixel 3a smartphone with an API level of 33 during testing. To ensure the breadth of testing the emulator was interacted with for 10-minute sessions to collect data.

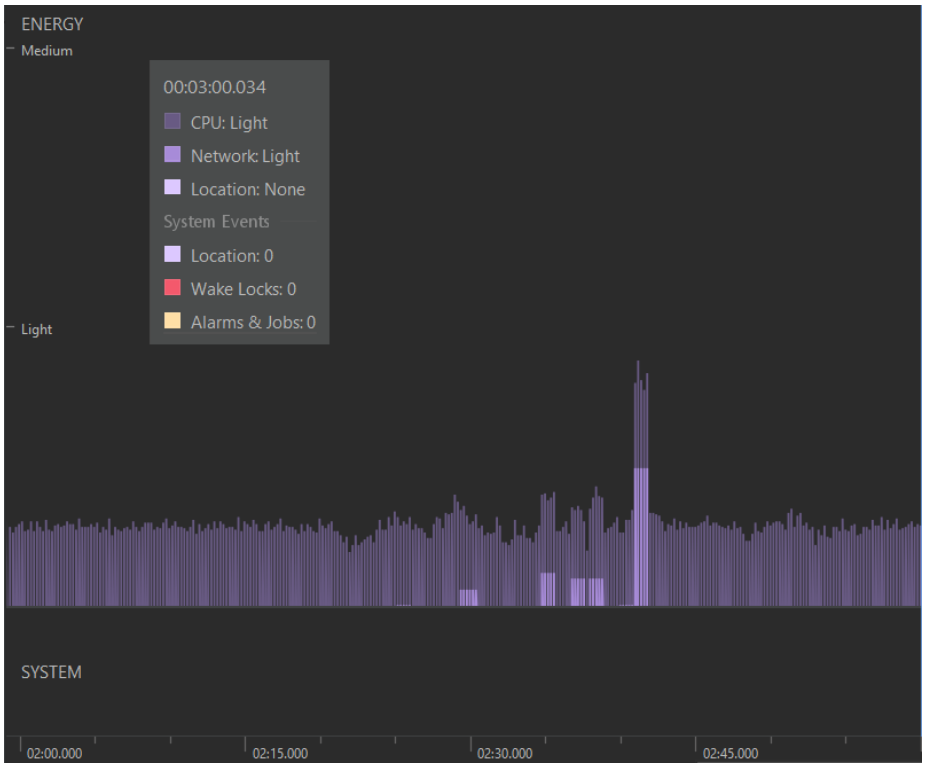

Figure 22: Example of Peak Energy Usage

Upon running Android Studio Profiler we found that the PrivacyMonitor module consumes a consistently low amount of the phone's battery life, as shown in the figure. The profiler gives information on CPU and memory usage as well. We consume on average 27 percent of the phone's CPU over a run and average use of about 186MB. In comparison, the Appjudicator module consumed on average 4 percent of the phone's CPU over a similar duration of time and intensity of interaction. The extra CPU usage is likely to be due to rendering for the bitmap representations of icons and general GUI improvements.

5.3 Privacy Metrics

There are two distinct approaches the system could employ to do its data collection and storage on the Corporate SDN. Their design, and how they affect the privacy of user-collected data, determines which approach meets our research goals. The synchronous table is designed using the schema shown in the Synchronous Updates table. Data collection is performed at timed intervals, in

Synchronous Updates				
UUID	BatchID	Policy Log	Hits	Date
A	1	Policy 1	X	mm/dd/yy
A	1	Policy 2	X	-
B	1	Policy 1	X	-
B	1	Policy 2	X	-
A	2	Policy 1	X	mm/dd/yy
A	2	Policy 2	X	-
B	2	Policy 1	X	-
B	2	Policy 2	X	-

Asynchronous Updates				
UUID	BatchID	Policy Log	Hits	Date
A	1	Policy 1	1	mm/dd/yy
B	1	Policy 2	1	-
B	1	Policy 1	1	-
B	2	Policy 2	1	-
A	1	Policy 1	1	-
A	2	Policy 2	1	-
B	2	Policy 1	1	-
B	2	Policy 2	1	-

Synchronous and Asynchronous Policy Reporting Examples.

which phones coordinate to send their full list of policies per batch. As we will discuss in our results section this design better meets privacy goals at the cost of some security value.

The key difference in the asynchronous system is batch id reporting and sending timings. With the table shown in Table 2, the data is less structured. The table is populated at more dynamic intervals. Each phone reports a policy hit as soon as it is generated. As such, the analysis of network flows in the system can happen quicker on the Corporate SDN side since data is immediately being transmitted. However, this table structure is subject to privacy concerns in the way data storage is handled. For example, the K anonymity value of the dataset is 1 when considering only the first four rows, since that is the only batch id with value 2 in its reporting. Yet, once the sixth row is considered the k value of the dataset increases to 2.

In order to measure our metrics we created python scripts that populated separate datasets which were subsequently analyzed. The systems report batch ids in a manner described in the Synchronous and Asynchronous sections. Further, the number of policies defined, the range of timestamps allowed, and

the number of phones simulated on the system are able to be customized in each script. An additional script was created to calculate K-anonymity and L-diversity using PyCanon [25].

5.3.1 Synchronous System Results

```
k-anonymity min: 5
The dataset verifies:
                - k-anonymity with k = 5
                - (alpha,k)-anonymity with alpha = 0.2 and k = 5
                - l-diversity with l = 5
                - entropy l-diversity with l = 5
                - (c,l)-diversity with c = 2 and l = 5
                - basic beta-likeness with beta = 0.0
                - enhanced beta-likeness with beta = 0.0
                - t-closeness with t = 0.0
                - delta-disclosure privacy with delta = 0.0
```

Figure 23: Privacy Calculations for Synchronous System (PyCanon Output)

According to our results, the Synchronous system had a K-anonymity and L-diversity score which always matched the number of phones being reported on as sensitive attributes. An example of the analysis performed on 5 phones with 7 policies and 2 date representations is shown in the figure. The system's theoretical design of having each phone report its batch, even when hits are 0, is what causes this behavior. Since per batch, each phone gives a description of all of its policies the quasi-identifiers will never make a combination that is unique within the batch. The batch numbers will always contain sets of identical information along with the date. Since each policy is represented in each batch there can be no unique row created that another phone did not also format the same way. We allow data collection to be set over a period of time to allow for batches to synchronize between phones. We say the Synchronous system is proven to be privacy-preserving based on our guidelines in the experimental setup since the re-identification risk is the ideal 1/k where k is the number of unique sensitive attributes reported on.

5.3.2 Asynchronous System Results

The Asynchronous System's generation of data is unstructured compared to the previous one. Each individual phone is sending tuples of data when packets are initially associated with policies. As such, the k-anonymity value will change as

new entries are inserted into the table. Since each phone is no longer responsible for ensuring its data is being stored in a privacy-preserving manner, we must describe how quasi-identifiers mathematically determine the K-Anonymity and L-diversity of the dataset at different times. Thus, we can model the K-anonymity value of a set of tuples grouped by batch id from each of the phone's total records classified with that batch id and given policy and the number of representations for a date.

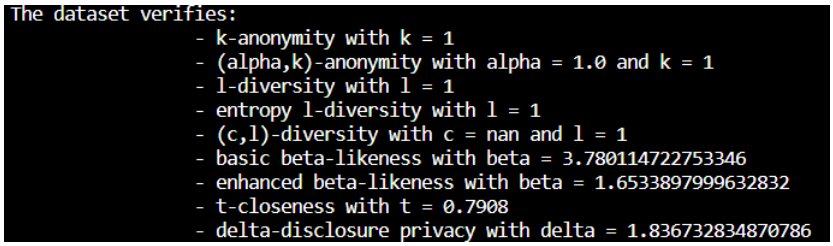

Figure 24: Privacy Calculations for Asynchronous System (PyCanon Output)

The results of our data creation configured with the same number of policy and date combinations lead to a K value of 1. At least one tuple in the dataset is wholly unique in its combination of quasi-identifiers. Since any records with a K-anonymity of 1 can be reasonably mapped to UUID, the dataset is not privacy-preserving. This is a consequence of the unavoidable real-life mapping between UUID and the phone that a particular instance of PrivacyMonitor is running on. Given a sizable amount of records per batch this issue may be alleviated somewhat. In practice, the testing of the system was too small in scope to prove this.

5.4 Testing TLSi

We verified the correctness of our implementation of the server-side implementation of TLS 1.2 against OpenSSL's industry-grade implementation of a TLS 1.2 client.

Since we ran into some reliability issues when running Appjudicator both with and without our modifications, we decided to go with a more controlled and restricted testing environment application which was more reliable while still capturing the essence of how Appjudicator was going to use TLSi.

Our testing application has two settings. The first setting is the default and it is supposed to represent how applications might establish an SSL session

with a remote server without any interference from Appjudicator. For this, we connected an SSLSocket object provided by the standard Java library internal to our testing application to a Python3 server running TLS on the machine hosting the emulated device. A representation of the default setup is shown in Figure 25.

We then proceeded to evaluate the performance impact of our implementation on the time it took the system to perform a TLS handshake and the time it took for the system to send a TLS payload to a server and receive its response.

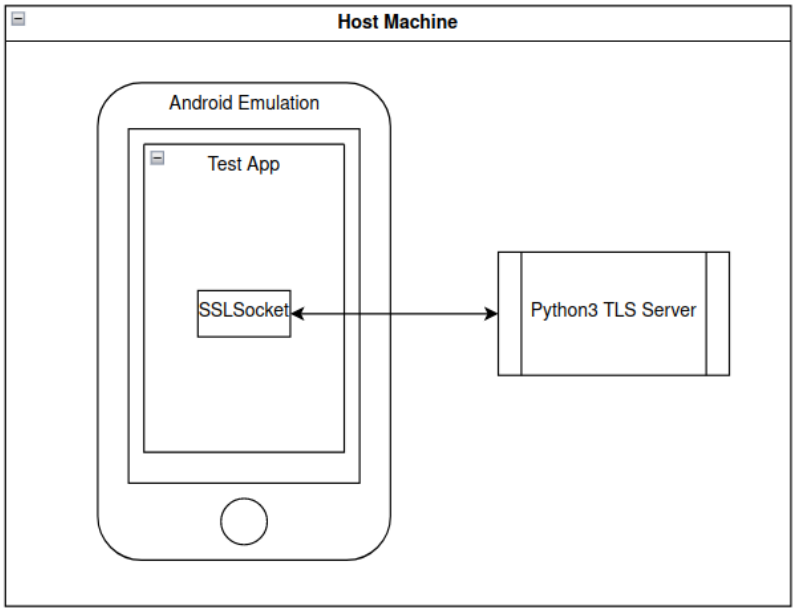

Figure 25: Default Performance Evaluation Setup

We then measured the time it takes the SSLSocket object to complete a TLS Handshake. Once the handshake is performed, the client sends an arbitrary message of random length (up to 1024 bytes) to the remote server. The remote server is set up to simply echo the same message it received back to the client. The client then verifies the messages sent and received match and records the time between sending and receipt of the message.

We named the second setting TLSi. This setting represents as closely as possible how our system is integrated inside Appjudicator. The representation is shown in Figure 26.

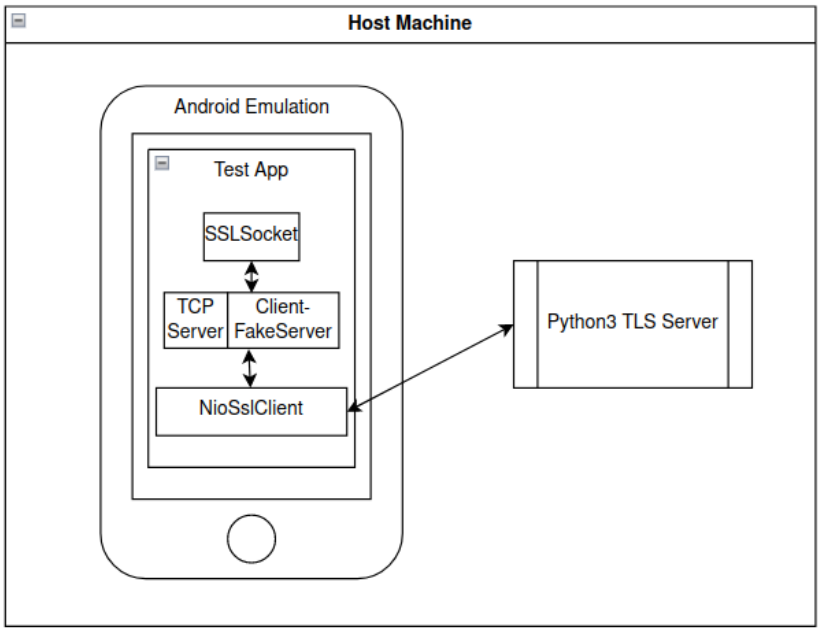

Figure 26: TLSI setting Performance Evaluation Setup

Just as with the default setting, the TLSi setting also has a client which uses an SSLSocket object. This client will eventually attempt to send a message using TLS 1.2 to the same remote Python3 TLS server running on the host machine. The difference now is that we simulate Appjudicator via 3 other sockets. The first socket is a TCP ServerSocket waiting to accept connections. Instead of connecting the client to the remote server, we instead instruct it to connect to our "fake server". This represents how Appjudicator uses the VPN service to capture packets. Upon accepting a connection we obtain a "Client-FakeServer" socket. Immediately after accepting a connection, our testing harness creates a NioSslClient object and connects it to the remote server the same way Appjudicator would with our integration of TLSi. Once that TLS session is established, Client-FakeServer then performs the TLS Handshake with the client. Once the handshake is complete, the time to handshake is then recorded. Similar to the prior setup, the client generates an arbitrary message with a random length (up to 1024 bytes) and sends it to the Client-FakeServer socket. Here we simulate Appjudicator by decrypting the packet using our server implementation of TLS 1.2. We then take the decrypted packet's payload and forward it to NioSslClient, which then re-encrypts and forwards it to the remote server. Upon receiving of

the response from the server at the NioSslClient, we re-encrypt the response using our implementation of the TLS Server and send the encrypted payload back to the client via the Client-FakeServer. Finally, the client receives the packet, decrypts its contents, and verifies that the message matches the original.

We do note that in a more natural setting, the client socket would be a part of another application and not within the testing application (which represents Appjudicator). We chose this setup instead because of easier development and less variability: here we only need to press 1 button in a single application; whereas in the other, more natural setting, we would have to start two applications and then switch between them when running the test, which could lead to varied results due to how Android performs its resource management when apps are running in the background. We note that the setup we used instead represents an ideal case when it comes to Androids resource management. We argue that, since the client Socket is inside the testing application in the Default and TLSi settings, the data we obtain will still result in an insightful comparison when evaluating performance.

We repeated this process 1000 times for both settings and recorded times for both the Time-To-Handshake (TTH) and the Round-Trip-Time (RTT). We used "SystemClock.elapsedRealtimeNanos()" to get a nanosecond precision for our time estimates.

5.4.1 Experimental Results

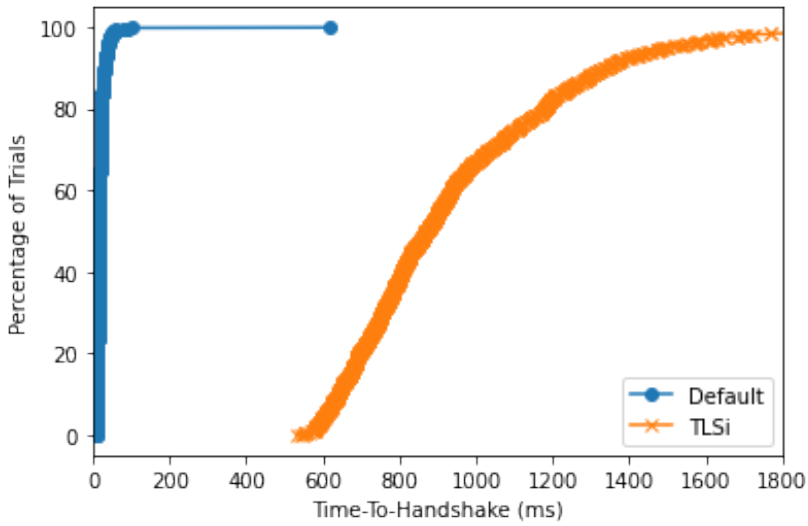

Figure 27: CDF of TTH for Default and TLSi Configurations

In the average case, we found that the TTH for the Default setting was around 19.47 milliseconds with a standard deviation of 21.10ms. The average TTH for the TLSi setting was 942.35 ms with a standard deviation of 292.87 ms. This is almost a 50-fold increase. The cumulative distribution graph of TTH for the different settings is plotted in Figure 27.

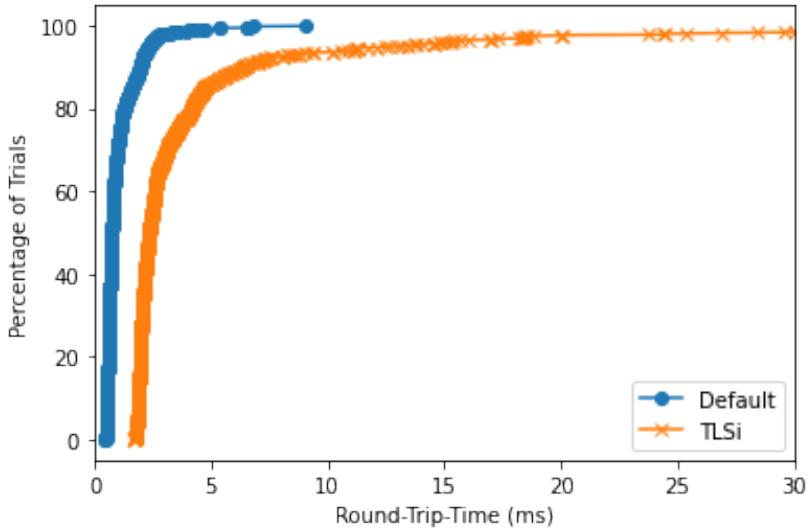

Figure 28: CDF of RTT for Default and TLSi Configurations

In the average case, we found that the RTT for the Default setting was around 1.02 ms, with a standard deviation of 0.77 ms. The average RTT for the TLSi setting was 4.22 ms, with a standard deviation of 7.13 ms. This is more than a 4-fold increase. Although the slowdown is significant, is much milder than for TTH. We show the RTT for the different settings plotted in Figure 28.

5.4.2 TLSi Results Evaluation

The results show that our implementation of TLSi clearly comes at a performance cost. The performance cost for performing when it comes to TTH is an order of magnitude greater compared to the performance cost of RTT which ensues after. However, we argue that since handshakes only have to be performed once at the start of a TLS session, as opposed to the TLS encrypted communication, this is a more favorable outcome than if it was the other way around.

What did surprise us is the actual values for the slowdowns themselves. We expected that the slowdowns for both TTH and RTT would be twofold. This is because, for the handshakes, instead of performing one TLS Handshake between client and remote, we would be performing two handshakes: one between client and Client-FakeServer, and the other between NioSslClient and remote. Similarly for RTT instead of encrypting a message at the client and sending it to the

server and then decrypting, and repeating the same process in the reverse order, now in the intermediate step we are decrypting the packet at Client-FakeServer, and re-encrypting and sending at NioSslClient.

Instead, we found slowdowns of almost 50 for TTH and more than 4 for RTT. We speculate that this comes down to us using Java and Kotlin for development, whereas the SSL Socket from the standard Java library and python's TLS Server both use binaries compiled in the C programming language. Additionally, we speculate that the bigger slowdown for TTH comes from the fact that we had to issue more Java operations and calls to the BouncyCastle library when handshaking relative to when performing encrypted communication post handshake.

While examining the data points, we noticed that for TTH in the Default setting the first trial consistently resulted in a TTH which is an order of magnitude greater than almost all TTH's in the remainder of the dataset. Similarly, the TTH for the first trial in the TLSi setting was consistently around 3 times greater than the remaining TTHs. This is shown in Figure 29.

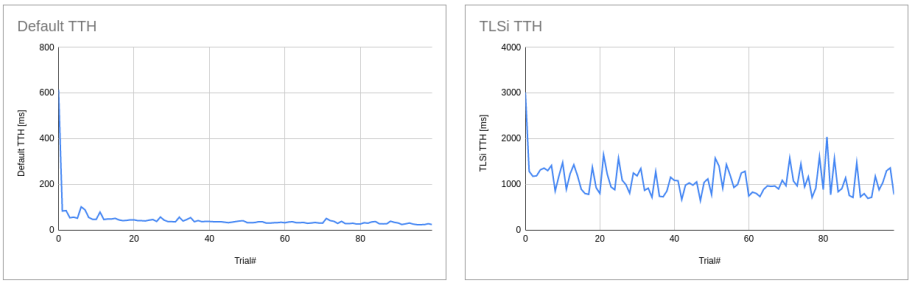

Figure 29: TTH Over Trials for Default and TLSi Configurations
Note: Y-axis scales are different

In the figure, we have only shown the first 100 trials for better clarity. The data seems to be distributed in a similar fashion for trials 10 up to 1000.

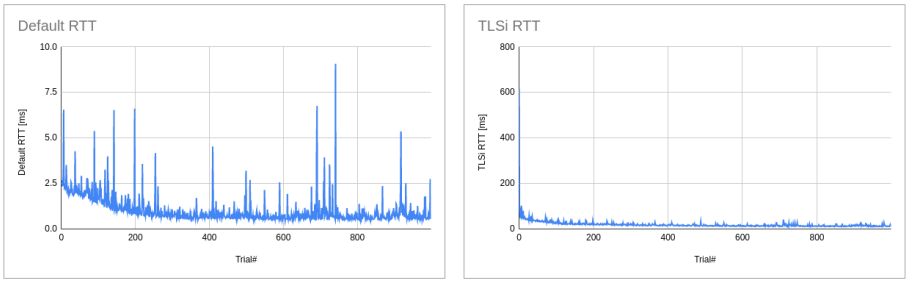

Figure 30: RTT Over Trials for Default and TLSi Configurations
Note: Y-axis scales are different

Additionally, we also noticed that the RTT trends downwards for both settings over time. We also plot this observation in Figure 30.

We therefore decided to exclude the first 10 trials from our initial estimates of the median and standard deviation. We argue that they are outliers to the dataset, and not including them yields a better representation of the average performance. With the first 10 trials excluded, the average TTH in the Default and TLSi settings is 18.43 ms and 937.86 ms, with standard deviations of 7.99 ms and 285.07 ms. This still results in a 50-fold slowdown in performance, but now the standard deviation for the Default setting has dropped almost 3-fold.

We would also like to point out that the TTH for the first Trial for the Default setting is around 615ms, which is fairly close to the average TTH of 935ms for the TLSi setting. We wondered whether the libraries which we used in the Default setting performed session resumption which would make the TTH for all the subsequent trials orders of magnitude smaller. However, using Wireshark to capture these handshake packets, it seems that no session resumption was performed in either setting. Since the TTH improves drastically after the first trial for both settings (not just the Default), we speculate that this is due to the system caching the different functions it needs to call in memory, leading to shorter TTHs and RTTs over time.

6 Discussion, Limitations, and Future Work

Here we outline Privacy Monitor's applications and discuss alternative approaches and solutions to the system's limitations. A review of the future work is then done.

6.1 Implications and Applications

We believe that PrivacyMonitor is an exciting idea that could have many applications in the space of MDM software. Our system provides a flexible framework that both preserves user privacy and allows the user to define their expectation of privacy, while simultaneously providing a sufficient level of control for a network operator. Our policies and effects of application classifications can be easily modified, providing a flexible system open to customization and improvement. PrivacyMonitor is a step in the direction of bridging the gap between invasive and under-secured MDM software.

6.2 Privacy Monitor's Limitations and Future Work

Although Privacy Monitor provides an informative and private system, it has significant limitations.

6.2.1 Batch System Policy Collisions

Due to time constraints, our team was not able to create a system capable of handling policy collisions. We define policy collisions as overlaps between matching criteria for policies. For example, the following network traffic could fall under both of the following policies:

Policy 1: IP destination: 192.168.xx.x — App classification: corporate

Policy 2: IP destination: 192.168.x.x

A network flow in this example could very easily fall under both policies and currently, Privacy Monitor has no way to handle such collisions. Our team recommends a priority system of some kind that would favor the most specific policy available.

6.2.2 Limited OpenFlow Table Modifications

Due to time constraints, our team was not able to implement complex changes to the flow table. Rather, our technical implementation proves that through the use of our external system, a network operator can create rules using our new protocol that will impact the modification of the device's FlowTable in some way.

Our team decided to use Appjudicator's existing system of consulting the off-device controller to modify the flow table for the sake of time, rather than creating a new method of updating the flow table. We do so by modifying the

"OFPT_PACKET_IN", which is then sent to the off-device controller, which makes its decision to update the flow table based on this packet.

In future work, our team suggests a system that uses our new protocol to make meaningful and complex changes to the flow table on the device.

6.2.3 Testing Limitations and Impact on Table Design

When designing our database models for running testing there were claims that we could not evaluate given the hardware we use and testing script implementation. Most importantly the data sets we generated are limited arbitrarily by computing power. Ideally, a simulation could be run with more resources to better describe a day's worth of data from a corporation. This would mean drastically increasing the number of policies, number of batches, and representations of dates we report on. Ideally, with large volumes of data coming in, the Asynchronous Design could be implemented as the database would be unlikely to ever be in a privacy-breaking state. As it stands the Asynchronous System will nearly always be privacy-breaking from our simulations.

Additionally, PrivacyMonitor can only support one phone connecting to our external system. This required us to collect and test data by artificially populating our tables, based on the example we had from the real population of our table with one phone.

6.3 Limitations, Privacy Impact and Future Work of TLSi

Due to time constraints, we were not able to implement certain features of TLS. Most notably, we haven't implemented the ability to resume TLS sessions, which would make re-handshaking much faster. We have also not implemented support for TLS 1.3. As discussed, TLS also provides extensions that change its behavior or provide additional data for the handshaking process. So far we have only partially implemented support for the renegotiation info and the EC point formats extension.

We also mentioned that it would be beneficial if Appjudicator would generate the self-signed root certificate and private key and then prompt the user to install it. However, whether a root CA installed in this manner can be used for our purposes for current Android versions is still unclear.

As alluded to in the Certificate Revocation subsection, one of the concerns with TLSi regards handling certificate revocations. Since our TLS Inspector initiates a new TLS handshake with the server, we need to manually check

whether the SSL certificate provided by the server was revoked or not. The original plan was to follow the method outlined by Taylor *et. al.* [16].

We leave the implementations of these features for future work.

6.3.1 TLSi Privacy Concerns

An additional concern regarding TLSi is the invasion of privacy that it allows. TLSi is often implemented in middleboxes for the purposes of security. In a corporate network, for example, employees might be uncomfortable with the fact that their company's middlebox breaks their HTTPS connection and inspects all of their data. Our solution avoids this concern by performing TLSi locally, on an employee's device. All traffic that Appjudicator then deems private to the employee will be encrypted back before it even reaches the company network. Using our batch system, we also protect the privacy of the employees by sending non-identifying packet information to the corporate logging service.

6.3.2 Alternative Failed Approach to TLS Inspection Implementation

Finally, we discuss a failed approach, which seemed very intuitive, but due to certain, yet unexplained intricacies of how Android implements its VPN Service, has failed.

As with any programming project, we would like to minimize the amount of unnecessary work by building upon using pre-existing code and libraries of others that were made publicly available. This was our original plan.

Initially, we re-imagined the workflow of Appjudicator, where for each new TLS flow, Appjudicator would create two sockets, a TLS server socket that would act as a "fake" server to which client packets are redirected, and a TLS client socket, which would act as a "fake" client which communicates with the remote endpoint. The setup would act very similarly to the "TLSi" setting of our TLSi testing.

Upon receiving a packet from the client app destined for a remote server, Appjudicator would overwrite the packet's destination IP and port, such that it matches the IP and port number of the fake server, and then write the packet out on its TUN interface. The fake server would receive the packet, and produce an appropriate response for both the TLS handshake and encrypted communication by using the SSLServerSocket [43] class from the Java standard library. This way, we would have avoided having to make our TLS packet parser as well as

our own implementation of the Server-Side TLS 1.2 Handshake. As a bonus, we would also have a TLS 1.3 implementation, as well as all future TLS versions the Java Library ends up implementing.

However, what we found was, although we could successfully modify and send these packets on the TUN interface, they would never be received by the SSLServerSocket. We could not find any documentation on why this might be the case. We speculate it is because the local TUN interface registers the socket to IP and port mappings on-the-fly as it receives packets. Since a server socket that only waits to accept connections never sends a packet until a connection is received, the TUN interface then never creates a socket to IP and port mapping for it, essentially making it accidentally drop all packets destined for our fake server.

6.3.3 TLSi: A better approach

Very late into development, during the testing phase, we realized that there does exist a standard library implementation of TLS which provides all of the functionality that we have manually implemented for TLSi. This is the SSLEngine class [2], which was featured in the borrowed code for NioSslClient. The documentation on the SSLEngine was a bit unclear, but essentially it allows the creation of a transport-layer-independent implementation of TLS, as TLS is usually carried via TCP. The SSL Engine parses and decrypts TLS packets, and it produces a raw TLS packet as a response to either a handshake message, or when encrypting application data. This raw data can then be sent via the chosen transport layer protocol.

We have noticed that the SSLEngine when performing a client-side TLS 1.2 Handshake needs to send an unnecessary message of all zeros so that its internal state transitions into a Handshake Finished. We then made some modifications so that this message is not sent, without impacting the SSLEngine. Without the in-depth knowledge needed to implement TLS 1.2 manually, we would have never caught this discrepancy, and we wouldn't have known how to fix it even if we did.

6.4 Conclusion

The Privacy Monitor module achieves its goal of preserving a user's privacy while simultaneously giving network administrators tools for threat detection and defense. Throughout the project, due to circumstances, concessions had to

be made, resulting in some systems being less robust than we had hoped. These limitations however allow for meaningful additions to be made to our project.

Our batch system preserves privacy through generalizing network activity such that sensitive information would not be revealed. Our policy issuance and traceback systems provide an additional layer of security for network operators. This balance allows the user to feel comfortable using the system while simultaneously allowing a network administrator to feel the network is secure

The Synchronous System was implemented into PrivacyMonitor as the Corporate SDN's storage and data retrieval model. It best met our privacy goals by accurately anonymizing the data in a privacy-preserving manner as reflected by its K-anonymity value. Since both table designs work with the SocketService and BatchCreation model of PrivacyMonitor we can say that our systems design and implementation extend OpenFlow capabilities to monitor employee-generated data using a Corporate SDN that preserves privacy while still achieving security goals through its controls.

Through our design, our team was able to create a system that has proven to preserve privacy while also providing a network operator with strong enough tools to protect a network. We were also able to extend the functionality of Appjudicator through the implementation of a TLS inspector or TLSi, although at a higher-than-expected performance cost.

References

[1] Christopher Allen and Tim Dierks. *The TLS Protocol Version 1.0*. RFC 2246. Jan. 1999. DOI: 10.17487/RFC2246. URL: https://www.rfc-editor.org/info/rfc2246.

[2] Yakov Rekhter and Kireeti Kompella. *Signalling Unnumbered Links in Resource ReSerVation Protocol - Traffic Engineering (RSVP-TE)*. RFC 3477. Feb. 2003. DOI: 10.17487/RFC3477. URL: https://www.rfc-editor.org/info/rfc3477.

[3] A. Barth et al. "Privacy and contextual integrity: Framework and applications". In: *2006 IEEE Symposium on Security and Privacy (Samp;P'06)* (2006). DOI: 10.1109/sp.2006.32.

[4] Tim Dierks and Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.1*. RFC 4346. Apr. 2006. DOI: 10.17487/RFC4346. URL: https://www.rfc-editor.org/info/rfc4346.

[5] A. Machanavajjhala et al. "L-diversity: Privacy beyond K-anonymity". In: *22nd International Conference on Data Engineering (ICDE'06)* (2006). DOI: 10.1109/icde.2006.1.

[6] Khaled El Emam and Fida Kamal Dankar. In: *Protecting Privacy Using k-Anonymity* (2008). DOI: 10.1197/jamia.M2716.

[7] Nick McKeown et al. "OpenFlow". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74. DOI: 10.1145/1355734.1355746.

[8] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/info/rfc5246.

[9] One Way et al. *Transport Layer Security (TLS) Renegotiation Indication Extension*. RFC 5746. Feb. 2010. DOI: 10.17487/RFC5746. URL: https://www.rfc-editor.org/info/rfc5746.

[10] Zhemin Yang et al. "Appintent". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer amp; communications security - CCS '13* (2013). DOI: 10.1145/2508859.2516676.

[11] Yabing Liu et al. “An End-to-End Measurement of Certificate Revocation in the Web’s PKI”. In: *Proceedings of the 2015 Internet Measurement Conference*. IMC ’15. Tokyo, Japan: Association for Computing Machinery, 2015, pp. 183–196. ISBN: 9781450338486. DOI: 10.1145/2815675.2815685. URL: https://doi.org/10.1145/2815675.2815685.

[12] Morufu Olalere. In: *A Review of Bring Your Own Device on Security Issues* (2015). DOI: 10.1177/2158244015580372.

[13] Wenfeng Xia et al. “A Survey on Software-Defined Networking”. In: *IEEE Communications Surveys and Tutorials* 17.1 (2015), pp. 27–51. DOI: 10.1109/COMST.2014.2330903.

[14] Wei Yang et al. “AppContext: Differentiating malicious and benign mobile app behaviors using context”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015). DOI: 10.1109/icse.2015.50.

[15] Kathleen Moriarty et al. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Nov. 2016. DOI: 10.17487/RFC8017. URL: https://www.rfc-editor.org/info/rfc8017.

[16] Curtis R. Taylor and Craig A. Shue. “Validating security protocols with cloud-based middleboxes”. In: *2016 IEEE Conference on Communications and Network Security (CNS)*. 2016, pp. 261–269. DOI: 10.1109/CNS.2016.7860493.

[17] Christopher Demicoli. *Never accept an MDM policy on your personal phone*. Jan. 2018. URL: https://blog.cdemi.io/never-accept-an-mdm-policy-on-your-personal-phone/.

[18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[19] CyberlinkASP. *Cloud hosting: Surprising byod stats and facts you should know*. 2020. URL: https://www.cyberlinkasp.com/insights/surprising-byod-stats-and-facts-you-should-know/.

[20] Ahmadreza Montazerolghaem. “Software-defined load-balanced data center: Design, implementation and performance analysis”. In: *Cluster Computing* 24.2 (2020), pp. 591–610. DOI: 10.1007/s10586-020-03134-x.

[21] Android. *The Android profiler android developers*. 2021. URL: https://developer.android.com/studio/profile/android-profiler.

[22] Apple. *Differential Privacy Overview - Apple Inc.*. 2021. URL: https://www.apple.com/privacy/docs/Differential_Privacy_Overview.pdf.

[23] Joseph Petitti. In: *Appjudicator: Enhancing Android Network Analysis through UI Monitoring* (May 2021).

[24] Craig McCart. *15 shocking BYOD statistics from 2018 - 2023*. Nov. 2022. URL: https://www.comparitech.com/blog/information-security/byod-statistics/.

[25] Judith S'ainz-Pardo D'iaz and 'Alvaro L'opez Garc'ia. "A Python library to check the level of anonymity of a dataset". In: *Scientific Data* 9.1 (2022), p. 785.

[26] Mahboobeh Dorafshanian and Mohamed Mejri. "Differential Privacy: Toward a better tuning of the privacy budget based on risk". In: *Proceedings of the 9th International Conference on Information Systems Security and Privacy* (2023). DOI: 10.5220/0011896600003405.

[27] Microsoft. *What is device management?* 2023. URL: https://learn.microsoft.com/en-us/mem/intune/fundamentals/what-is-device-management#choose-the-device-management-solution-thats-right-for-you.

[28] alkarn. *Server and Client implementation with SSLEngine*. Accessed: 3-21-2023.

[29] OpenSSL Project Authors. *OpenSSl Home Page*. Accessed: 3-21-2023.

[30] Legion of the Bouncy Castle Inc. *BouncyCastle Home Page*. Accessed: 3-21-2023.

[31] Let's Encrypt. *Let's Encrypt Stats*. Accessed: 3-21-2023.

[32] Open Networking Foundation. *OpenFlow switch specification - open networking foundation*. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.0.0.pdf.

[33] Wireshark Foundation. *Wireshark Home Page*. Accessed: 3-21-2023.

[34] Android Developers Google. *Package Manager*. Accessed: 3-21-2023.

[35] Android Developers Google. *VPN*. Accessed: 3-21-2023.

[36] Cybersecurity Insiders. *BYOD Security Report*. Accessed: 3-21-2023.

[37] Borislav Kiprin. *What Is the Heartbleed Bug and How to Prevent It*. Accessed: 3-21-2023.

[38] Danny Palmer. *Over 400 instances of Dresscode malware found on Google Play store, say researchers*. Accessed: 3-21-2023.

[39] Pcap4J.org. *Pcap4J Home Page*. Accessed: 3-21-2023.

[40] Java™ Platform. *Standard Edition 7 API Specification, (2020). Class Signature*. Accessed: 3-21-2023.

[41] Java™ Platform. *Standard Edition 7 API Specification, (2020). Class Socket*. Accessed: 3-21-2023.

[42] Java™ Platform. *Standard Edition 7 API Specification, (2020). Class SocketChannel*. Accessed: 3-21-2023.

[43] Java™ Platform. *Standard Edition 7 API Specification, (2020). Class SSLServerSocket*. Accessed: 3-21-2023.

[44] Java™ Platform. *Standard Edition 7 API Specification, (2020). Class SSLSocket*. Accessed: 3-21-2023.

[45] Michael Cobb Rahul Awati. *certificate revocation list (CRL)*. Accessed: 3-21-2023.

[46] Jai Vijayan. *'Unkillable' Android Malware App Continues to Infect Devices Worldwide*. Accessed: 3-21-2023.