Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Software y Sistemas

Trabajo Fin de Máster

In-depth analysis of the Android supply chain:Vendor customizations on critical networking components

Author(a): Rathnayaka Mudiyanselage Vinuri Gayanthika Bandara
Tutor(a): Dr. Srdjan Matic
Tutor(b): Dr. Narseo Vallina-Rodriguez

Madrid, May 2023

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Máster Universitario en Software y Sistemas

Título: In-depth analysis of the Android supply chain:Vendor customizations on critical networking components

May 2023

Autor(a): Rathnayaka Mudiyanselage Vinuri Gayanthika Bandara

Tutor(a): Dr. Srdjan Matic
IMDEA Software Institute
ETSI Informáticos
Universidad Politécnica de Madrid

Tutor(b): Dr. Narseo Vallina-Rodriguez
IMDEA Networks Institute/AppCensus
Madrid

Resumen

La amplitud y extensión del proyecto de código abierto Android (AOSP) permiten a los vendedores de dispositivos Android (también conocidos como fabricantes de equipos originales) introducir personalizaciones en sus productos para diferenciarse en el mercado y añadir nuevas capacidades. Sin embargo, estas personalizaciones pueden tener implicaciones importantes y graves para la seguridad y la privacidad de los usuarios.

Los riesgos para la seguridad y la privacidad causados por la falta de control sobre la cadena de suministro de Android han llamado la atención de los investigadores en ciberseguridad. Estudios anteriores se han centrado en analizar los problemas de seguridad relacionados con las aplicaciones preinstaladas y las modificaciones realizadas en la tienda raíz de Android o en las configuraciones de red. A pesar de ello, existe una importante laguna en la investigación acerca de cómo las personalizaciones de los proveedores en la pila de red de Android pueden obstaculizar el establecimiento de comunicaciones de red seguras.

Para evaluar las amenazas a la comunicación segura introducidas por los vendedores, estudio las personalizaciones en la pila del protocolo TLS/SSL. Empleo técnicas avanzadas de análisis estático, concretamente diffing sobre datos de firmware de Android recopilados a través de campañas de crowdsourcing. Aplicando mi pipeline de análisis estático sobre un conjunto de datos de 48.520 dispositivos de más de 300 proveedores, detecto y analizo las desviaciones de los proveedores con respecto al proyecto oficial Android Open Source Project (AOSP), mantenido por Google. Al analizar las personalizaciones identificadas, descubro vulnerabilidades de seguridad críticas que pueden comprometer la seguridad de los usuarios y de las aplicaciones. Éstas van desde malas prácticas de los proveedores, el uso de versiones antiguas de la plataforma Android, parches de seguridad críticos retrasados, implementaciones criptográficas obsoletas, distribuciones inseguras de proveedores criptográficos como versiones vulnerables de OpenSSL, hasta la ausencia de funciones de seguridad avanzadas como validación de certificados, verificación de nombres de host y cifrados priorizados debido a la eliminación por parte de los proveedores de métodos públicos estándar que ofrecen estas capacidades.

En particular, estas deficiencias persisten tanto en los proveedores certificados de Android como en los no certificados. Esto sugiere una falta total de control sobre la cadena de suministro y su cumplimiento de las mejores prácticas que afectan directamente a los intentos de los desarrolladores de aplicaciones de proteger sus aplicaciones utilizando la pila de protocolos nativos. Los resultados preliminares de esta tesis ponen de manifiesto la necesidad de controles más estrictos sobre la cadena de suministro de Android. De hecho, creo que los reguladores y las autoridades

de certificación pueden promover nuevas iniciativas para reforzar las garantías de seguridad de los dispositivos y controlar las prácticas de los distintos actores de la cadena de suministro de Android.

Abstract

The openness and extensibility of the Android Open Source Project (AOSP) enable Android device vendors (also known as Original Equipment Manufacturers) to introduce customizations in their products for market differentiation and adding new capabilities. However, these customizations can have significant and severe implications for user's security and privacy.

The security and privacy risks caused by the lack of control over the Android supply chain have caught the attention of cybersecurity researchers. Previous studies have focused on analyzing the security issues related to pre-installed applications and modifications made to the Android root store or network configurations. However, a significant research gap exists due to the lack of investigation into how vendor customizations on Android's network stack can hinder the establishment of secure network communications.

To assess the threats to secure communication introduced by vendors, I study the customizations on the TLS/SSL protocol stack. I employ advanced static analysis techniques, specifically diffing on Android firmware data gathered through crowdsourcing campaigns. By applying my static analysis pipeline over a dataset of 48,520 devices from more than 300 vendors, I detect and analyze vendor's deviations from the official Android Open Source Project (AOSP), maintained by Google. By analyzing the identified customizations, I uncover critical security vulnerabilities that can compromise users' and application's security. These range from poor vendor practices such as using older Android platform releases, delayed critical security patches, outdated cryptographic implementations, insecure distributions of cryptographic providers like vulnerable versions of OpenSSL to the absence of advanced security functions such as certificate validation, hostname verification, and prioritized ciphersuites due to vendors' removal of standard public methods offering these capabilities.

Notably these shortcoming are persistent both within Android certified vendors as well as non-certified ones. This suggests a total lack of control over the supply chain and their compliance with best practices that directly impact on app developers' attempts to secure their applications using the native protocol stack. The preliminary findings reported in this dissertation, highlight the need for stricter controls over the Android supply chain. In fact, I believe that regulators and certification authorities can promote new initiatives to strengthen device security guarantees and control the practices of the different actors in the Android supply chain.

Contents

List of abbreviations

AOSP	Android open source project
CDD	Compatibility definition document
CTS	Compatibility test suite
.dex	Dalvik executable
.jar	Java archive
JCA	Java cryptography architecture
JCE	Java cryptography extension
JSSE	Java secure socket extension
.odex	Optimized dalvik executable
.oat	Optimized android file format
ODM	Original design manufacturing
OEM	Original equipment manufacturer
OS	Operating system
.so	Shared object
SNI	Server name indication

Chapter 1

Introduction

As of today, Android is the dominating mobile operating system (OS) with a 78% market share.[1] One of its keys to success is its open source nature, which is maintained by Google under the name **Android Open Source Project (AOSP)** [14]. However, its open source nature has enabled a large and complex software supply chain which includes stakeholders such as:

- Chipset/processor manufacturers like Qualcomm or MediaTek.
- Original Equipment Manufacturers(OEMs) such as Google, Samsung, HTC and Huawei, which are the phone vendors considered by buyers when purchasing an Android device. Certain vendors may sell their products under different brands.
- Mobile network operators (MNOs), such as Telefonica or Orange, may preloaded their own software in subsidized Android devices.
- Android app markets such as Google Play, the Baidu app store [18], or the Amazon App Store [1] may come preloaded in Android devices. When an Android vendor is Google-certified [7] may also come with other Google's products such as Gmail, Google Maps, or Youtube.
- Third-party applications like social networks, browsers, streaming platforms, games or anti-virus software that could be bundled as a preloaded app when an Android device is released to the market.

AOSP can be heavily modified by any of these actors to incorporate additional features that can differentiate their products from other vendors or to pre-install third-party applications such as MNO apps for billing, browsers, or social network apps. The feature additions done on the AOSP could extend from adding vendor-unique preinstalled apps (also known as preloaded applications), broadening the usability of the original applications and overall creating an exclusive look and character for their Android devices. However, while the booming number of feature addition on Android devices is attractive from an end-user perspective and also from a revenue perspective, this raises concerns regarding the privacy and security guarantees of these products as reported by prior work: customizations and pre-installed apps are privileged system components that cannot be removed without rooting the device [69]. Moreover,

[1] iOS vs Android Quarterly Market Share

actors in the supply chain may also introduce privacy-intrusive third-party advertising and tracking libraries in their apps, which can silently harvest users' personal data without their awareness and consent.

It is, therefore, pivotal to understand the nature and risks associated with OS-level customizations, their origin and their purpose. I consider two types of customizations:

- Customized Android core components that come packaged with any Android device as they are a critical part of the Android Operating System (e.g., the Telephony Manager, Contacts Manager, or the Location Manager). These elements are initially developed and maintained by the development team behind the AOSP but vendors can add or remove features, and even include third-party tracking libraries on them for monitoring users.
- Vendor core components, which are structural elements that come preloaded in an Android framework but are are created and embedded by different actors in the supply chain (e.g,. a browser, app store, or music player). These elements presents the most potential of being different between vendors due to the supply chain relationship and product diversity; additionally the vendor-defined core components can also be specific to a region due to MNO operations or chipset manufacturers or region-specific requirements.

The escalating security and privacy risks associated with the lack of control over the Android supply chain have increasingly come under the scrutiny of the cybersecurity researcher community. I discuss extensively prior work in Section 2. Yet, prior research has been dedicated to analyzing the security implications of customizing pre-installed apps, specifically (i) investigating the issues of over-privileged applications, misuse of third-party libraries and permissions [69], and (ii) compliance issues among vendors [87]. A separate but complementary line of research has focused on identifying the vulnerabilities that arise due to modifications made by supply chain actors to the Android root store [93, 82], or to network configurations [92, 83, 73].

1.1 Knowledge gap and Dissertation Objectives

A significant but critical research gap remains unexplored in the research literature: **how customization and source code alterations on Android's network stack can impair the establishment of secure network communications**. Vendor customizations in the SSL/TLS network stack —both at the kernel and Android framework level—can affect the establishment of secure communications. Inadequate customizations done upon the SSL/TLS stack could expose the end-user to unforeseen threats due to insecure communication channels such as state surveillance or in-path proxies. With the goal of narrowing this research gap, this dissertation has the following two goals:

1. Develop scalable and accurate methods to identify vendor customizations on the Android's SSL/TLS networking libraries. Specifically, we focus on customizations on the:
    - Android Java secure socket extension API implementation
    - Android native cryptographic providers

- Android Java cryptographic Extensions

2. Examine how the lack of control over the Android supply chain is responsible for platform fragmentation, potential compatibility issues (i.e., by removing standard API features), and even for impairing end-users' networking security by introducing vulnerabilities.

The customizations on the Android native cryptographic providers and the Java cryptographic extensions ultimately contribute to the networking abilities of an device through Android Java secure socket extension APIs. These APIs are critical from a network security standpoint as they determine the default configuration for SSL/TLS communication and allow the developers to integrate enhanced security practices such as TLS based protocols [50], setting prioritized ciphersuites [51], certificate pinning [53, 52].

1.2 Contributions

Using a novel purpose-built static analysis method (Described in Chapter 4), this dissertation empirically analyzes for the first time (i) the extent of vendor's customizations on critical networking components on Android devices, and (ii) assess to which extent they could become threat to end-user's security. By applying my analysis pipeline on a dataset formed by over 48,500 Android images, spanning across the latest 5 versions and covering 300 different vendors, we find that:

- **Level of customizations.** I analyzed the Android devices released under 300 vendors running on Android version 9, 10, 11, 12 and 13 to characterize the level of customizations done on the Android SSL/TLS stack by vendors through differential analysis (i.e., "diffing"). My analysis on the Java secure socket API vendor implementations shows that around 70% of the vendor versions show deviations from the established baseline which is the AOSP, with some even lacking basic endpoint verification functionalities implemented by OpenJDK. The analysis on the Java cryptographic providers, provided insight into the low-level provider shifts done by vendors resulting in OpenSSL and it's alternative derivatives (Libgcrypt) being used as the cryptographic primitive and protocol provider instead of BoringSSL, the default cryptographic provider by Google in response to OpenSSL vulnerabilities. Finally the examination on Java cryptographic extension layer (default Conscrypt as used by Google AOSP) showed that some vendors do not give access to the latest cryptographic implementations to be used in the application level. For example, I found instances where specific ciphersuites introduced for the advancements in TLS 1.3 are lacking in SSL/TLS stacks of some Chinese device vendors.
- **Security issues due to customizations.** The analysis done on each SSL/TLS critical component showed that developer lack discipline and best practice approaches when implementing and maintaining Android networking capabilities, therefore falling behind on basic end-user security. The usage of OpenSSL alternatively to BoringSSL, raises doubts on the vendor knowledge on the best effort approaches in the community. I found isolated cases where, the version of BoringSSL used within the device was potentially released two years prior to the device release therefore showing the importance of proper validation of such critical components and bad praxis by supply chain actors. The examination of

all three critical component layers, suggested a lag in vendors' diligence to incorporate the latest security efforts taken by Google. Samsung devices showed a delay in deprecating a patchy Diffie-Hellman protocol by one year and Xiaomi devices are yet to incorporate hybrid public key encryption to their networking capacities by the year 2021. These findings are more worrying given that both Samsung and Xiaomi are Google certified vendors according to Google's list [7]. The missing functionalities catering to certificate verification found within the analysis of Java cryptographic extensions, can potentially impair user and app security by limiting application developers' ability to incorporate additional security measures such as certificate pinning.

- The explorations conducted on vendor SSL/TLS stacks, within the scope of the research objectives lead to to identification of commonality among the vendors. On the network interface level, some vendors showed a staggering similarity between their API specifications. Yet, I observe methods for certificate transparency and verification missing in a certain set of Android vendors, many of which are not certified by Google. Despite the complex nature of the Android supply chain and the lack of transparency of their Android manufacturing processes, reaching a conclusion on the actual reasons behind these findings is difficult. Yet these results could lead to the identification of some mutual stakeholders providing Android images to different brands, as well as common insecure practices by some actors in the Android supply chain.

All in all, this study reveals the shortcomings and the probable scope for improvements in Android vendor TLS/SSL implementations. I conclude this dissertation with a general discussion of the findings and a discussion of potential mitigations that could rectify the inconsistencies and the poor security development practices observed in the Android supply chain, including independent certification efforts and stricter regulatory compliance in the context of the new EU Cyber Resilience Act [26].

Chapter 2

Background

This chapter discusses the technical background knowledge required to understand the scope of this dissertation. First I introduce the basic structure of the Android open source project (Section 2.1). Then, I present the fundamentals components of Android architecture (Section 2.2). Lastly, I describe the networking libraries, with an emphasis on SSL/TLS, used to establish network communications (Section 2.3).

2.1 Android Open Source Project (AOSP)

The Android open source Project [14], led and maintained by Google, serves as the basis of the different Android variants found in the market. AOSP is a complete and well-maintained project with thorough documentation which provides Android vendors with almost all the functionalities to create a functional Android device, either a smartphone or a Smart TV. Yet, its open source nature allows vendors to customize AOSP by modifying the code, selecting dependencies (e.g., cryptographic providers) or integrating third-party apps to meet their specific needs and differentiate their products from competitors. The overall impact and process of the supply chain is presented in Section 3.1.

The Android supply chain initiates from the need to included the additional components such as kernel drivers with respect to each custom Android variant. This phenomena, known as *"fragmentation"*, is responsible for producing a myriad of Android variants that may have their own vulnerabilities and threats, and for potential incompatibilities by modifying the native APIs that developers can invoke. When it comes to custom Android variations, the closest derivation of the AOSP is found with in Android Pixel phones [37], which are manufactured by Google itself.

```
. AOSP/
    - art
    - bionic
    - bootable
    - build
    - compatibility
    - cts
    - dalvik
    - developers
```

```
- development
- device
- external
- frameworks
- hardware
- kernel
- libcore
- libnativehelper
- Makefile
- out
- packages
- pdk
- prebuilts
- sdk
- system
- tools
- vendor
```

2.2 The Android OS Architecture

The Android open-source platform is a complete software stack that includes multiple components such as the operating system, middleware and system applications. Android is built primarily using Java programming language but some low-level libraries are implemented using C/C++. It consists of multiple components stacked and grouped into different architecture layers, as shown in figure 2.1 [15]:

- **Linux Kernel.** This the the underlying base structure of the Android architecture, the ART utilizes this layer for low level functionalities such as memory management and multi-threading. The underlying kernel is different for each Android version release since this is modified by Google using the already existing Linux open-source OS.
- **Hardware Abstraction Layer (HAL).** The hardware abstraction layer (HAL) is an abstraction layer built upon the Android linux kernel which includes a set of standard interfaces manufactured or implemented by hardware manufacturers or or the chipset manufacturers as we refer to them in the Android supply chain. This layer allows the programs to seamlessly communicate with the kernel layer components like the camera driver even if substantial changes are done on the Android OS.
- **Android Runtime (ART).** The ART establishes the runtime environment for running Android applications and system services using the strategy called "Ahead-of-Time (AOT)" compilation. This turns the complete application code into native bytecode (`.dex` representations) during the installation and stores it for the time being.
- **Native C/C++ libraries.** Native C/C++ libraries play a crucial role in improving Android performance along with application functionality. Their purpose could ranges from database connection support, cryptographic primitive implementations to TLS protocol implementations. Since the ART is built using native code,

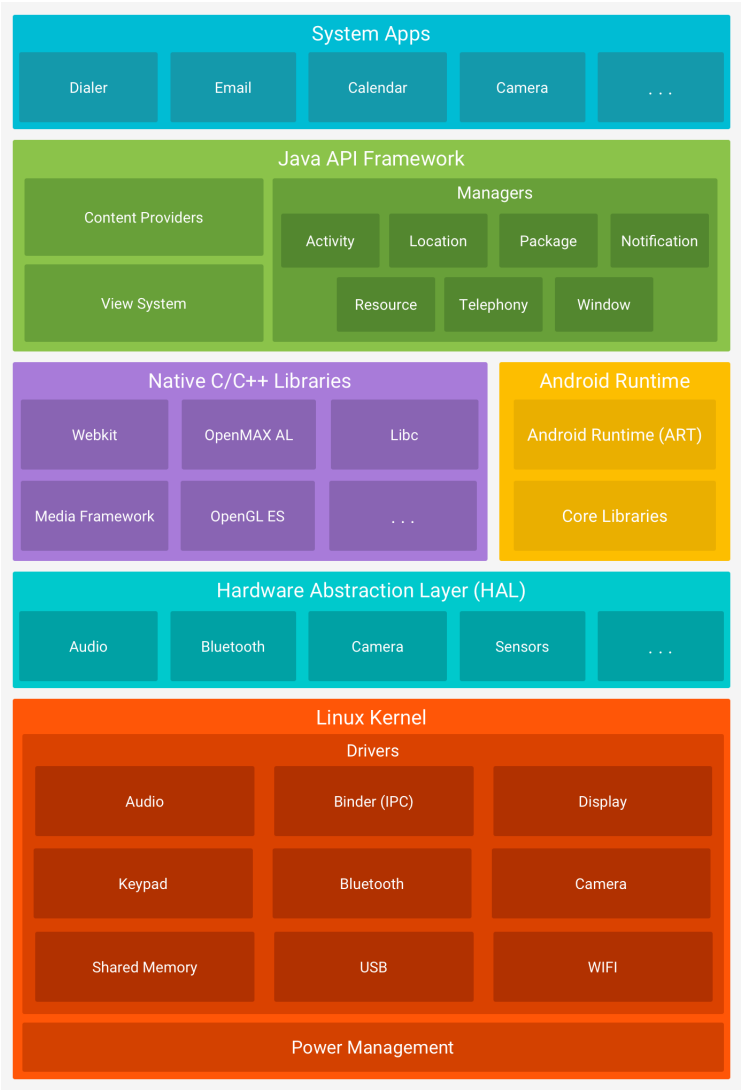


Figure 2.1: Android software stack (source : `https://developer.android.com/guide/platform`).

the required libraries by the Android OS are also implemented using native code (C or C++). The developers can access system-level functionalities provided by these libraries through implementations of Java APIs that bridges the gap between the native layer and the high-level user layer. This integration between the Java code and the native layer is facilitated by the Java native interface (JNI) and allows developers to incorporate features from both Java-based platform and the native implementations.

For example the cryptographic primitives and TLS implementations which are incorporated as native libraries are exposed to the Android application layer through Java APIs which are constructed through Conscrypt which is an Java wrapper around the networking native libraries. During the Android build process, the C/C++ sources of these libraries are compiled into ELF binary files and are stored in the system partition. Therefore this allows the vendor/OEM to include their own libraries or extra libraries that support the inherent func-

tionalities or advanced functionalities for their applications and hardware.

- **Android Framework.** This layer allows user-space applications to access lower-layer components through the JAVA API framework. The application programming interfaces (APIs) made available to developers through the framework layer gives them an opportunity to create applications using higher-level languages (e.g., Java) and yet communicate with the more advanced and complicated Android OS using the simple APIs. Even though this layer creates easy access points for developers to communicate with Android OS, some of the APIs might allow access to sensitive data hardware components if not regulated properly. Therefore the Android architecture includes a permission-based security mechanism which requires the user to accept or decline the usage of these sensitive APIs. When using an Android application when requires the user location, the Android OS will require user permission to allow access to such sensitive information.
- **Application Layer.** The top layer is the application layer which contains both user-space Android applications as well as preinstalled ones (typically installed in the `system` partition). User-installed applications are typically done through any Android market place or through an Android Package (APK) installation (i.e., side-loading). On the other hand the system apps are embedded in the device's system partition and are typically included when creating the image, yet they can be added or modified during the updates/security updates performed by Firmware-Over-the-Air (FOTA) components [63], which are typically under the control of the original OEM/vendor but can be externalized to companies like Redstone. Due to the read-only nature of the systems partition, users simply can't uninstall preinstalled apps, which also run with privileged system privileges.

2.2.1 Building an Android Image

When building an Android build using the AOSP code or a modified version, the output will be stored in the `Out/` folder which then can be flashed into a physical handset using Google's platform tools such as the Android SDK platform tools [16]. An Android image is typically structured as follows:

1. art : Compiles the latest Android runtime (ART) environment.
2. bionic : Contains Android's C library, math library , dynamic liner interfaces (as a library) [20].
3. bootable : OS startup or boot related code is available in this folder.
4. build : The starting point to building the Android OS is through the build/envsetup.sh. This prepares the build environment by ensuring all the dependencies are fulfilled.
5. compatibility : Testing and helper components for ensuring the compatibility between different Android versions and with the Android platform. Includes critical Google initiatives such as the Android compatibility definition (CDD) [5].
6. cts : Contains the test suites as required as CDD, which is namely the Android compatibility test suite (CTS).

7. dalvik : Source code responsible for establishing the Dalvik virtual machine for Android devices.
8. development : Aids the development by providing source for tools such as ndk and sdk.
9. device : Some common product specific configurations that can be used for different devices. Such as the Google Android accessories kit and template for creating a customized Android shared library along with how to embed in the Android device using the Java Native interface(JNI).
10. external : Supports the extensibility of the AOSP. This folder includes the source code from all the external open source projects used when building the Android version; such as Webkit, Boringssl, Apache Http, OkHttp etc.
11. frameworks : The source for the Android framework. Containing the implementations for system services such as package and activity managers along with the methods of communication between the Java APIs and the native libraries.
12. hardware : The HAL related source code such as hardware specifications and implementations.
13. kernel : specific to the devices, includes the linux kernel source code based on the device configuration setup.
14. libcore : Includes source for ojluni, luni, libart which are supporting libraries for core functionalities in the Android OS [39].
15. libnativehelper : Contains a set of JNI utilities to be used in Android OS implementation.
16. packages : The Android standard applications that are core components such as the telephone, camera.
17. pdk : Includes the platform development kit,to be used when creating custom Android powered devices. This helps the manufacturers to migrate to newer releases since this contains the necessary components for implementing the HAL.
18. prebuilts : Binary forms of necessary packages and libraries used for successfully building Android OS.
19. sdk : Some applications which are not part of the Android OS, these can be further built upon by the developers or used when creating custom applications.
20. system : Source code for the core Android system. The base linux system before the Dalvik virtual machines and Java services are added on top. Mainly the init process and script are included.
21. tools : External tools aiding the build and compilation of the OS.
22. vendor : vendor specific libraries which are most of the time proprietary and non-open source.

All of these sections listed above, are integral parts of the Android OS and during the build process can be extended according to vendor requirements. For example the external section can extended by adding external open source packages such

as LibreSSL and any vendor-specific implementations that are closed source can be included within the vendor partition during the

Customizations shouldn't cause incompatibility issues with the standard Android applications. To ensure this compatibility of customized devices, Google initiated the CDD and the CTS. Any vendor who wants to market their device as an Android device can use the CTS which is a set of tests to assure the AOSP compatibility. While the CTS is open source there are other test suites used by Google as way of clearing the Android vendor as a Google certified vendor [7]. Vendors who do not adhere to these tests are not allowed to come bundled with Google's app suite, yet they can include Google Play by outsourcing their devices to a certified original design manufacturers or ODMs [7]. Google Play is the most popular app store to install third-party applications on the device, currently hosting over 1M apps. Yet, users and vendors can opt to use alternative app stores like Xiaomi's or Baidu's [94].

2.3 The SSL/TLS Protocol Stack

The Android ecosystem provides security to its users by protecting the data transfers that enters and leaves the device, this is accomplished through transport layer security (TLS) which is the predecessor of secure socket layer (SSL) which are the two mostly used cryptographic protocols. From Android 10 onwards TLS 1.3 is enabled by default for all the TLS connections being established, while TLS 1.2 was supported since Android 12.

HTTPS (Hypertext Transfer Protocol Secure) utilizes the TLS to encrypt data moving back and forth between client devices and servers, which is controlled and maintained through the android:usersCleartextTraffic configuration in an Android application. This attribute is set to "false" by default as a best-effort method for preventing clear text traffic from an application. In Android the trust model of TLS is based on the TLS certificates presented in the Android root store, which will be further discussed in section 2.3.2, and the technical implementations and the lower level details on the TLS/SSL implementation in Android context will be discussed in section 2.3.1.

2.3.1 Android TLS/SSL implementation

The openness and extensibility of the AOSP is not only limited to upper layer levels such as the Android applications but also escalates to lower level implementations such as TLS protocol implementations. While the usability of these implementation narrows down to establishing a secure connection, the underlying architecture is rather complicated and needs to be studied extensively. The given figure 2.2 shows these critical components, adhering to more abstract concepts these components can be grouped into two classes; JSSE (Java secure socket extension) API and the JCA (Java cryptographic architecture) which compiles into the Android cryptographic software stack. This stack is implemented using Java (JSSE, OkHttp, Conscrypt) and C (Boringssl), while during the build process the C/C++ libraries are compiled and stored as shared library files or `.so` files and the Java libraries/source code is compiled and stored as `jar`, `oat`, or `odex` files depending on the Android build and version [9].

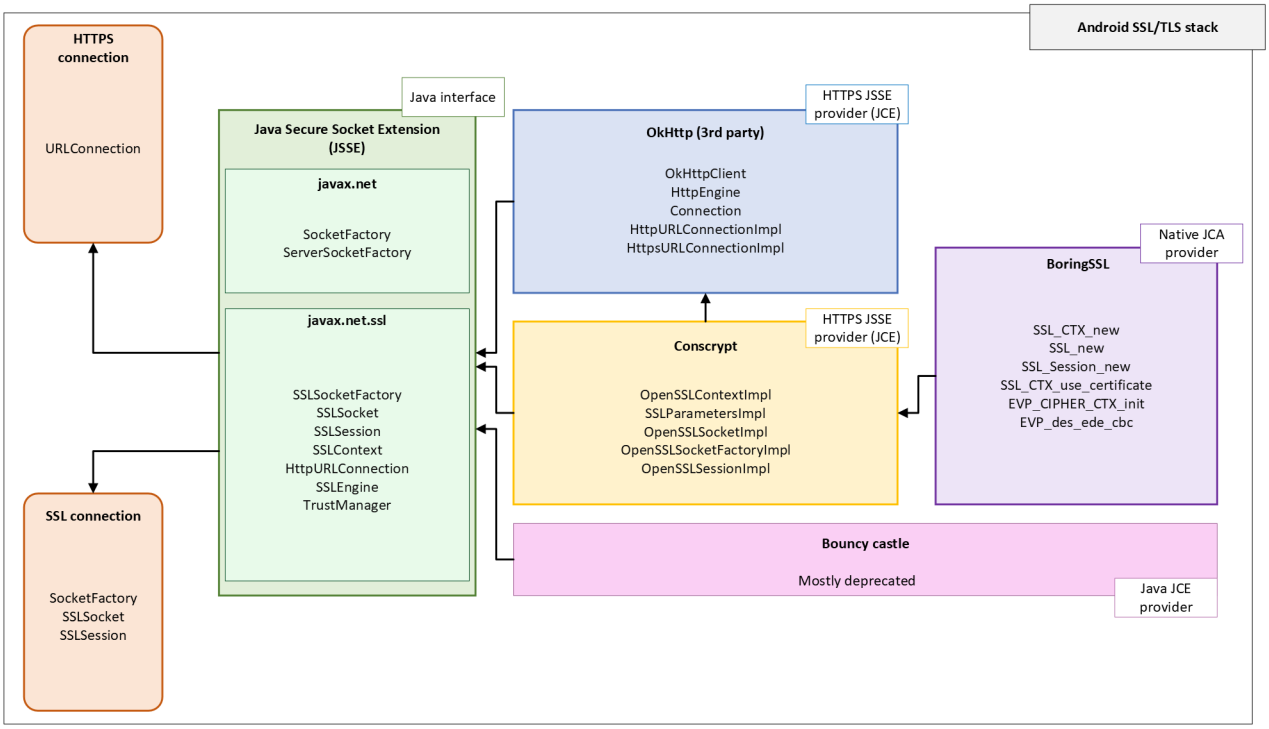


Figure 2.2: Android HTTPS/TLS implementation and core components.

2.3.1.1 Java Secure Socket Extension (JSSE) API

The Java Secure Socket Extension (JSSE) is the layer between the mobile application/service and the JCA/HTTPS providers, this bridges the gap between the native libraries layer and the Android application layer. The two main classes which hold support for Android TLS/SSL are `javax.net` [12] and `javax.net.ssl` [13]. These classes provide groundwork for services such as ssl client, server socket, ssl engine (supports the I/O operations of the application), ssl context , https url connection, trust management and key management.

Table 2.1 provides an overview of the important `javax.net` and `javax.net.ssl` subclasses, along with their purpose in the SSL/TLS context. Most of the functionalities defined in these classes supplements the core networking and cryptographic functionalities defined in `java.net` [10] and`java.security` [11] packages.

To observe how `javax.net.ssl` extends other classes such as java.* , org.*, the following figure 2.3 depicts the high-level connection between the javax and java packages.

The javax API access the low level implementations of secure communication protocols offered by "providers" through Conscrypt which acts as the bridge between the native BoringSSL libraries and the high-level Java interfaces. As shown in figure 2.2 `javax.net.ssl` and `javax.net` only includes interfaces, the reciprocal classes are implemented in Conscrypt and BoringSSL. For example, through figure 2.2 we can create the internal dependencies for `javax.net.ssl.SSLContext`; while Conscrypt's OpenSSLContextImpl and SSLParametersImpl classes create the relationship between Java classes and C libraries ,BoringSSL's Struct_SSL_CTX provides the actual TLS context.

JSSE class	JSSE subclass	Purpose
javax.net	ServerSocketFactory	Create server sockets
	SocketFactory	Create sockets
javax.net.ssl	CertPathTrustManagerParameters	Handles certificate path parameters
	ExtendedSSLSession	Extension of the SSL session interface to add additional attributes such as supported signature algorithms
	HandshakeCompletedEvent	Indication of the completed SSL handshake
	HttpsURLConnection	Extend java.net.HttpURLConnection with support for HTTPS features
	KeyManagerFactory	Factory for key managers based on key store or provider
	KeyManagerFactorySpi	Service provider interfaces for the above KeyManagerFactory class
	KeyStoreBuilderParameters	X509KeyManager parameter object encapsulates the list of KeyStore builders
	SNIHostName	Create instances to represent DNS hostname in a Server Name Indication extension
	SNIMatcher	Create instances to represent a matcher for performing match operations on an SNIServerName instance
	SNIServerName	Create instances to represent a server name in a Server Name Indication extension
	SSLContext	Represent a secure socket protocol implementation (The protocol could be a factory for secure socket factories and SSL engines)
	SSLContextSpi	Service Provider Interface(SPI) for SSLContext
	SSLEngine	Enable secure communication using protocols such as SSL and TLS
	SSLEngineResult	Encapsulate the resulting state produced by an SSLEngine I/O call
	SSLParameters	Encapsulates parameters for an SSL/TLS connection instance
	SSLPermission	This is noted as Legacy at the time of the research, recommended not to be used
	SSLServerSocket	Extend the ServerSockets and exposes secure server sockets using TLS or SSL protocols
	SSLServerSocketFactory	Create SSLServerSockets
	SSLSessionBindingEvent	Send the event to SSLSessionBindingListner; which is an interface for objects to ensure their SSL session status (bound or unbound)
	SSLSocket	Extension of Sockets for providing secure sockets using protocols, currently SSL and TLS
	SSLSocketFactory	Create SSLSockets
	StandardConstants	Include standard constants definitions
	TrustManagerFactory	Factory for trust managers based on the type of of trust material used by secure sockets. Currently supports the PKIX algorithm.
	TrustManagerFactorySpi	Define the service interface for the TrustManagerFactory class
	X509ExtendedKeyManager	Extension of the X509KeyManager interface
	X509ExtendedTrustManager	Extensions to the X509TrustManager interface to support sensitive trust management for TLS or SSL

Table 2.1: The JSSE API subclasses and functionalities [12, 13].

The internal supply chain of Google depends on OpenJDK [48] to embed the JSSE APIs into Android OS, these are pulled into `/platform/libcore/ojluni` and modified according to the needs of Android. The complication of keeping track of the changes done by each vendor for OpenJDK based files comes from the developer freedom to depend on different versions of OpenJDK, for example the following upstream versions of `javax.net.ssl` classes (as shown in figure 2.4) are extracted from different versions.

2.3.1.2 Java Cryptography Providers (JCA providers)

The Android architecture includes an extensible cryptographic and protocol provider stack, in the AOSP this is implemented using BoringSSL (derivative of OpenSSL) and Conscrypt (Java wrapper around native BoringSSL). From a developer stand point these provide the Java equivalents of BoringSSL functionalities available to developers in the forms of protocol implementations and cryptographic primitives.

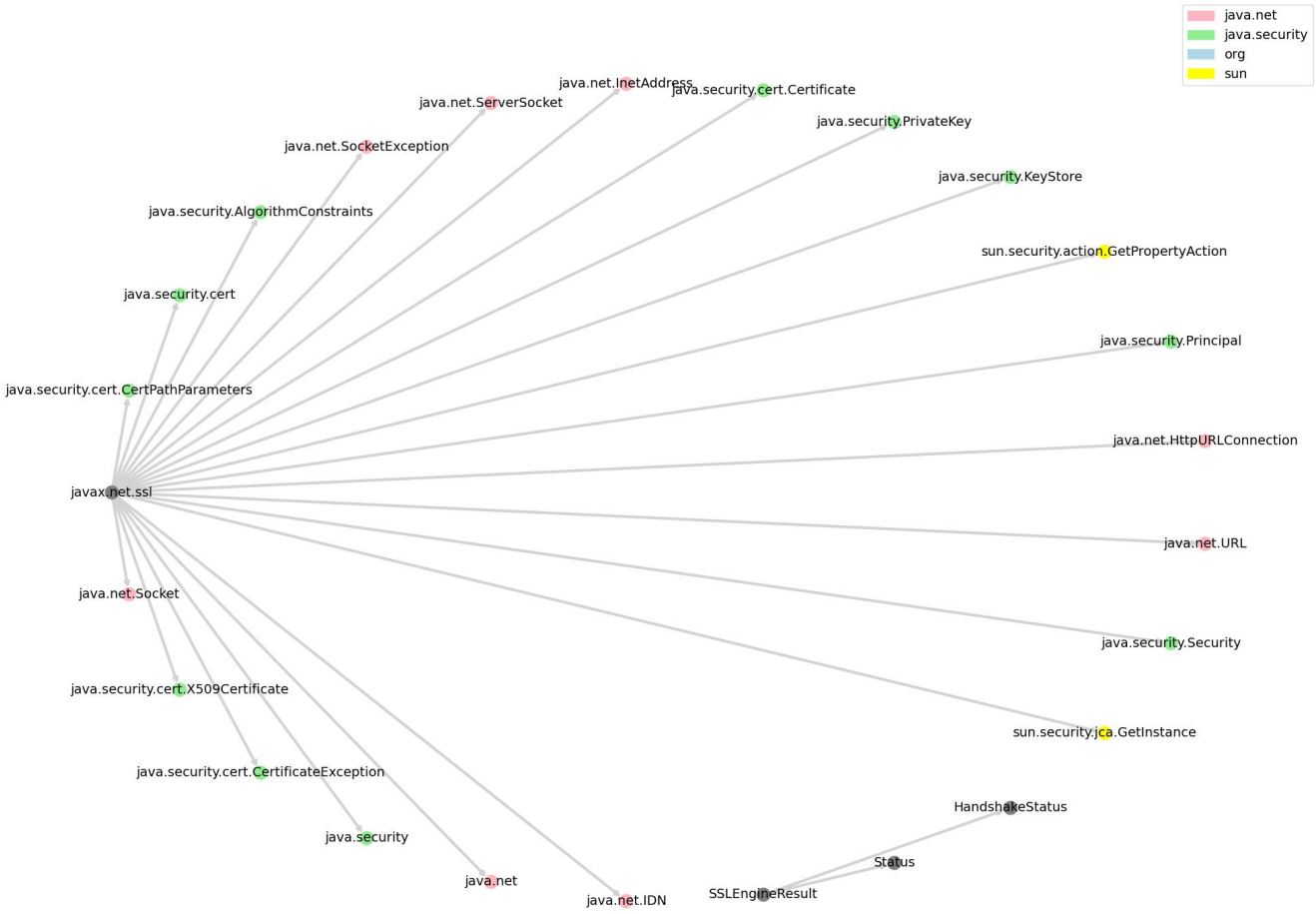


Figure 2.3: javax.net.ssl dependencies among java.* , org.* and sun.* classes.

BoringSSL

This low level native library provides the actual TLS functionalities to the Android OS. According to the Google source of BoringSSL [35] this is Google's fork of OpenSSL [85] and therefore is not intended to be used for external projects due to the API and ABI instability. Due to the high severity vulnerabilities [91] found in OpenSSL, and Google who was depending on OpenSSL until year 2014 decided to create their own fork due to high amount of patches OpenSSL released due to the exposed threats. Due to the non maintainability issues that arose within this period, Android shifted to BoringSSL leaving their OpenSSL [49] branch unmaintained. At the time of composing this document, this branch records it's last commit 7 years ago (In 2016). BoringSSL provides the basic primitives and TLS/SSL implementations and the chromium documentation [21] provides the overall functionality overview. When referring to the SSL implementation included in BoringSSL, the following protocols are currently available to be used within the Android OS.

```
#define DTLS1_VERSION_MAJOR 0xfe
#define SSL3_VERSION_MAJOR 0x03
#define SSL3_VERSION 0x0300
#define TLS1_VERSION 0x0301
#define TLS1_1_VERSION 0x0302
#define TLS1_2_VERSION 0x0303
```

```
ojluni/src/main/java/javax/net/ServerSocketFactory.java,jdk17u/jdk-17.0.6-ga,src/java.base/share/classes/javax/net/ServerSocketFactory.java
ojluni/src/main/java/javax/net/SocketFactory.java,jdk17u/jdk-17.0.6-ga,src/java.base/share/classes/javax/net/SocketFactory.java
ojluni/src/main/java/javax/net/ssl/CertPathTrustManagerParameters.java,jdk17u/jdk-17.0.6-ga,src/java.base/share/classes/javax/net/ssl/CertPathTrustManagerParameters.java
ojluni/src/main/java/javax/net/ssl/ExtendedSSLSession.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/ExtendedSSLSession.java
ojluni/src/main/java/javax/net/ssl/HandshakeCompletedEvent.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/HandshakeCompletedEvent.java
ojluni/src/main/java/javax/net/ssl/HandshakeCompletedListener.java,jdk17u/jdk-17.0.6-ga,src/java.base/share/classes/javax/net/ssl/HandshakeCompletedListener.java
ojluni/src/main/java/javax/net/ssl/HostnameVerifier.java,jdk17u/jdk-17.0.6-ga,src/java.base/share/classes/javax/net/ssl/HostnameVerifier.java
ojluni/src/main/java/javax/net/ssl/HttpsURLConnection.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/HttpsURLConnection.java
ojluni/src/main/java/javax/net/ssl/KeyManager.java,jdk17u/jdk-17.0.6-ga,src/java.base/share/classes/javax/net/ssl/KeyManager.java
ojluni/src/main/java/javax/net/ssl/KeyManagerFactory.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/KeyManagerFactory.java
ojluni/src/main/java/javax/net/ssl/KeyManagerFactorySpi.java,jdk11u/jdk-11.0.13-ga,src/java.base/share/classes/javax/net/ssl/KeyManagerFactorySpi.java
ojluni/src/main/java/javax/net/ssl/KeyStoreBuilderParameters.java,jdk17u/jdk-17.0.6-ga,src/java.base/share/classes/javax/net/ssl/KeyStoreBuilderParameters.java
ojluni/src/main/java/javax/net/ssl/ManagerFactoryParameters.java,jdk17u/jdk-17.0.6-ga,src/java.base/share/classes/javax/net/ssl/ManagerFactoryParameters.java
ojluni/src/main/java/javax/net/ssl/SNIHostName.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SNIHostName.java
ojluni/src/main/java/javax/net/ssl/SNIMatcher.java,jdk17u/jdk-17.0.6-ga,src/java.base/share/classes/javax/net/ssl/SNIMatcher.java
ojluni/src/main/java/javax/net/ssl/SNIServerName.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SNIServerName.java
ojluni/src/main/java/javax/net/ssl/SSLContext.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLContext.java
ojluni/src/main/java/javax/net/ssl/SSLContextSpi.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLContextSpi.java
ojluni/src/main/java/javax/net/ssl/SSLEngine.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLEngine.java
ojluni/src/main/java/javax/net/ssl/SSLEngineResult.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLEngineResult.java
ojluni/src/main/java/javax/net/ssl/SSLException.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLException.java
ojluni/src/main/java/javax/net/ssl/SSLHandshakeException.java,jdk11u/jdk-11.0.13-ga,src/java.base/share/classes/javax/net/ssl/SSLHandshakeException.java
ojluni/src/main/java/javax/net/ssl/SSLKeyException.java,jdk11u/jdk-11.0.13-ga,src/java.base/share/classes/javax/net/ssl/SSLKeyException.java
ojluni/src/main/java/javax/net/ssl/SSLParameters.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLParameters.java
ojluni/src/main/java/javax/net/ssl/SSLPeerUnverifiedException.java,jdk11u/jdk-11.0.13-ga,src/java.base/share/classes/javax/net/ssl/SSLPeerUnverifiedException.java
ojluni/src/main/java/javax/net/ssl/SSLPermission.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLPermission.java
ojluni/src/main/java/javax/net/ssl/SSLProtocolException.java,jdk11u/jdk-11.0.13-ga,src/java.base/share/classes/javax/net/ssl/SSLProtocolException.java
ojluni/src/main/java/javax/net/ssl/SSLServerSocket.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLServerSocket.java
ojluni/src/main/java/javax/net/ssl/SSLServerSocketFactory.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLServerSocketFactory.java
ojluni/src/main/java/javax/net/ssl/SSLSession.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLSession.java
ojluni/src/main/java/javax/net/ssl/SSLSessionBindingEvent.java,jdk11u/jdk-11.0.13-ga,src/java.base/share/classes/javax/net/ssl/SSLSessionBindingEvent.java
ojluni/src/main/java/javax/net/ssl/SSLSessionBindingListener.java,jdk17u/jdk-17.0.6-ga,src/java.base/share/classes/javax/net/ssl/SSLSessionBindingListener.java
ojluni/src/main/java/javax/net/ssl/SSLSessionContext.java,jdk11u/jdk-11.0.13-ga,src/java.base/share/classes/javax/net/ssl/SSLSessionContext.java
ojluni/src/main/java/javax/net/ssl/SSLSocket.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLSocket.java
ojluni/src/main/java/javax/net/ssl/SSLSocketFactory.java,jdk8u/jdk8u121-b13,jdk/src/share/classes/javax/net/ssl/SSLSocketFactory.java
```

Figure 2.4: Libcore upstream OpenJDK dependencies and the versions.

```
#define TLS1_3_VERSION 0x0304
#define DTLS1_VERSION 0xfeff
#define DTLS1_2_VERSION 0xfefd
```

When it comes to cipher suites, currently these algorithms have global functions that can be called from the java interfaces.

```
OPENSSL_EXPORT const EVP_CIPHER *EVP_rc4(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_des_cbc(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_des_ecb(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_des_ede(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_des_ede3(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_des_ede_cbc(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_des_ede3_cbc(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_aes_128_ecb(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_aes_128_cbc(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_aes_128_ctr(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_aes_128_ofb(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_aes_256_ecb(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_aes_256_cbc(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_aes_256_ctr(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_aes_256_ofb(void);
OPENSSL_EXPORT const EVP_CIPHER *EVP_aes_256_xts(void);
```

This library is included in the AOSP within the external/boringssl and within the build process is stored lib(lib64)/libssl.so and lib(lib64)/libcrypto.so (ELF representations) which respectively refers to TLS/SSL implementations and cryptographic primitives. This fork of BoringSSL is validated and certified through the "Cryptographic Module Validation Program" (certificate 3753) [46] to provide secure cipher suites and protocols.

Alternate crypto providers

The extensibility and the openness of the Android ecosystem allows the developers to embed their own cryptography providers into the native library layer. Following are some of the most popular cryptography providers developers can integrate into their TLS/SSL stack.

OpenSSL [85] was Android's choice of sole crypto provider until the year 2014 when OpenSSL's HeartBleed vulnerability which allowed anyone on the internet to read the system memory of any device/system that such integrated with the vulnerable version of OpenSSL. This issue brought upon many security patches that the AOSP had to send to it's devices. Due to the scale of remediation to be done and the following compatibility issues Google decided to fork their own OpenSSL branch as rename it to BoringSSL which is described in section 2.3.1.2. But due the extensibility and the openness of the Android eco system the developers still has the freedom to use any alternative over the recommended and comparatively vulnerability free BoringSSL.(The only CVE entry for BoringSSL was recorded in 2018 [24]). The current Android repository for OpenSSL has been left unmaintained. OpenSSL has obtained FIPS 140-2 validation which assures that specific security requirements for cryptographic modules have been satisfied [27].
GnuTLS [34] is a library for secure communication that provides developers with simplistic APIs implemented in C with access to protocols such as SSL, TLS and DTLS protocols with strong encryption capabilities such as AES and Camellia. GnuTLS claims that this provides smooth integration along with the rest of linux native libraries.
WolfSSL [95] is an attractive alternative for traditional TLS libraries given the small size , better memory utilization. The WolfSSL stack is made from two core components; the WolfSSl SSL/TLS library and the WolfCrypt crypto-engine. Their crypto-engine has received the FIPS 140-2 certification and the development team behind the library claims that this is a stable and progressive addition or alternative for long-time SSL/TLS implementations such as OpenSSL.
Botan [54] cryptographic library is implemented using C++ and is distributed under the permissive BSD license which gives developers the opportunity to use unmodified source or binaries of Botan in their implementations; this could reduce the attractiveness of Botan for Android developers.
LibreSSL [43] is a fork of OpenSSL which was made with the goal of security improvements , best development practices. Same as OpenSSL and BoringSSL , LibreSSL also includes libcrypto and libssl but also libtls which is an improved TLS library. libtls can be built upon libcrypto from LibreSSL, OpenSSL or BoringSSL.
Libgcrypt [33] created by GNU privacy guard is a general purpose cryptographic library which includes all the basic components for a cryptographic stack such as ciphers, hash algorithms, public key algorithms and MACs etc.

2.3.1.3 Java Cryptographic Extensions (JCE)

The Android JCE layer consists of Conscrypt and BouncyCastle as for the current AOSP source code. Among these two, Conscrypt is considered the primary security provider for the JSSE API level since the release of Android 8.0 [4]. Conscrypt can be completely replaced by third-party implementations such as BouncyCastle (which was the default security provider for the application level APIs until Android 8.0),

although it is mostly deprecated.

Conscrypt

Conscrypt which is implemented in Java, presents the Android developer with a correspondent to BoringSSL. Therefore is referred to as the Java cryptographic extension (JCE) [23] and is mainly exposed through the `javax.crypto` JSSE API. The following mapping can be used to further clarify the one-to-one mapping between Conscrypt and BoringSSL.

Conscrypt		**BoringSSL**
SSLParameterImpl	—>	SSL_CTX_new
OpenSSLSocketImpl	—>	SSL_new
OpenSSLSessionImpl	—>	SSL_SESSION_new

According to Android claims, incorporating Conscrypt guarantees an improvement of security without relying on the Over-the-Air updates. Within an Android device, the library is stored as a JAR files or OAT files depending on the version and configurations. From a secure communication point of view, Conscrypt provides support for the following TLS protocol versions.

- SSLv3 (Almost deprecated / If initiated, ignored by Android OS)
- TLSv1
- TLSv1.1
- TLSv1.2
- TLSv1.3

This also exposes implementations of different cipher suites which are already defined in BoringSSL , based on the required TLS protocol version (As shown in table 2.2).

BouncyCastle

BouncyCastle is a cryptography API provider for Java (JCE) and is minimally used for it's cryptographic algorithms by Android. BouncyCastle's implementations can be used in place of BoringSSL and Conscrypt, in terms of it's purpose. Due to security considerations, namespace conflicts BouncyCastle has two main releases used within the Android history.

First alternative is 'SpongyCastle' [57], a customized version of the original package to provides easier updates and android-specific functionalities. This distribution has not been updated since the year 2018 as the namespace conflicts behind the origin of SpongyCastle was resolved [19]. The second alternative was initiated due to the security concerns arose regarding the BouncyCastle implementation, this resulted in the development team behind BouncyCastle to develop StripyCastle. StripyCastle is a FIPS 140-2 release version of BouncyCastle, that only includes cryptographic modules that have been tested against the FIPS 140-2(Federal Information Processing Standard) therefore providing developers with a stronger addition to the JCE stack. According to the BouncyCastle documentation StripyCastle only supports upto Android Oreo (version 8).

TLS 1.0	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_RSA_WITH_AES_128_CBC_SHA TLS_RSA_WITH_AES_128_GCM_SHA256 TLS_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_256_GCM_SHA384
TLS 1.1	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_RSA_WITH_AES_128_CBC_SHA TLS_RSA_WITH_AES_128_GCM_SHA256 TLS_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_256_GCM_SHA384
TLS 1.2	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 TLS_RSA_WITH_AES_128_CBC_SHA TLS_RSA_WITH_AES_128_GCM_SHA256 TLS_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_256_GCM_SHA384
TLS 1.3	TLS_AES_128_GCM_SHA256 TLS_AES_256_GCM_SHA384 TLS_CHACHA20_POLY1305_SHA256

Table 2.2: Enabled cipher suites based on TLS protocol version

2.3.1.4 Android HTTPS Providers

OkHttp [90] developed by Square, is a third party HTTP/HTTPS and apache client provider in Android architecture. Although OkHttp is included in the Android AOSP SSL/TLS stack, it can be replaced by alternative HTTPS providers such as Volley [38], Retrofit [58] and Apache HttpClient [17]. In the context of HTTPS, the HttpsURLConnectionImpl of OkHttp supplies JSSE's HttpsURLConnection interface with a concrete class implementation; which mean OkHttp has the ability to provide HTTP/HTTPS connections and apache client interfaces by extending the java socket layer without relying on other dependencies. Similar to Conscrypt, OkHttp also exists in the external/okhttp within AOSP and after building it's stored as JAR files or OAT files.

2.3.2 Android Root Store

The Android root store plays a critical part in the establishment of secure communication, this is the Android storage for trusted root certificates. In the event of an Android device initiates an SSL/TLS connection with a server, it needs to perform certificate verification due to the standards of public key infrastructure (PKI). In summary, during this process the target server presents the digital certificate which holds it's public key. Then the client (in this case, the Android device) needs to verify the authenticity of the server certificate before establishing a secure communication.

For this purpose, the Android root store, located in `system/etc/securty/cacerts/` holds a collection of preloaded trusted root certificates issued by various certificate authorities (CAs). In a user stand point, during the verification process the Android SSL/TLS process checks whether the certificate sent by server is signed by a trusted CA. If so then a connection is established otherwise the connection is rejected or the Android user is prompted to make a decision on continuing with an untrusted connection. For Android version 13, the AOSP included a total of 129 trusted CAs in it's root store but this can be modified by vendors to add their own custom CAs to ensure the device's secure communication.

The JSSE holds the TrustManager which holds the responsibility of managing overall trust materials along with digital certificates and therefore utilizes the Android root store. BoringSSL holds the base functionalities such as SSL_CTX_set_verify_depth, SSL_CTX_set_cert_store.

The root store in Android devices are updated through FOTA updates to include new trusted root certificates or remove certificates that are no longer considered secure. (deprecated or vulnerable)

2.4 Compiling the Android source code

The Android source code documentation provides vendors with extensive instructions on how to build Android from the AOSP source code [6]. This requires downloading the AOSP source, initiating the build environment, selecting Android build variant and then finally building the OS. These steps are further discussed below.

1. Downloading the AOSP source

   This step requires the use of the tool 'Repo' [56]. Repo is an executable python script which can be incorporated in the target location of thr AOSP. This tool is built upon Git which helps with maintaining versioned code repositories. During building the AOSP, repo will be used to checkout and sync the source code.

```
1     $ curl https://storage.googleapis.com/git-repo-downloads/repo > <repo script path>
2     $ chmod a+x <repo script path>
3
4     Add repo to PATH variable
5     $ PATH = <repo script path>:$PATH
```

   As a pre-requirement, the developer should set up their git details in order to source the AOSP code.

```
1     $ git config --global user.name <Git username>
```

```
2     $ git config --global user.email <Git user email>
```

After the above steps are completed the developer can use repo, to initialie the required Android version to be built. The list of source code builds [8] has been made available to Android developers, therefore each AOSP platform release can be checked out and built using repo.

For example if the vendor device is built on Android build ID sp1a.210812.016.a1, that indicates that this device was built upon Android 12 platform release 3 (android-12.0.0_r3). Developers can initialize this platform release using the repo tool in a desired target location. Before downloading the AOSP source the space requirement of the build process should be taken into consideration. The AOSP source code requires 250 GB of free space and in order to build it, the space requirement is 150 GB.

```
1     # Using repo tool, the developer can check out android-12.0.0\_r3 release.
2     $ repo init -u https://android.googlesource.com/platform/manifest -b android-12.0.0\
          _r3
3
4     # After successfully checking out and initializing the repository, repo can be used
          to download the source code into a desired folder.
5     $ repo sync (should be executed within the desired target location)
```

2. Integrating vendor-specific code, packages and libraries into the AOSP source code.

   Add the vendor-specific code, packages and libraries into the appropriate locations within the AOSP source code. If these are proprietary packages or libraries, these should be addded in `vendor/` folder and any other open source implementations can be added to the `external/` folder. This can involve adding new components, modifying existing components, or creating custom implementations to extend the functionality of the Android system. After the additons of the vendor-specific modules, it's pivotal to resolve any dependency issues.

3. Setting up build environment

   For a successful build, the developer should initialize the execution environment using the envsetup.sh script provided by the AOSP.

```
1     # This should be executed at the root of the AOSP
2     $ source build/envsetup.sh
```

4. Choosing a target and a variant to build

   This allows the developers to select the product target which decides the features that are allowed on the device and the capability of the OS to run on different hardwares. This step also includes selecting the variant, which decides the behavior of the OS. For example if the variant 'eng' is selected then the OS will include additonal debugging capabilities.

```
1   # Launching the lunch command allows the developer to select the target and the build
        variant.
2   $ lunch
```

When the above command is executed, the following choices will be displayed to the developer.

```
vinurib@hydra:~/AOSP$ lunch

You're building on Linux

Lunch menu .. Here are the common combinations:
     1. aosp_arm-eng
     2. aosp_arm64-eng
     3. aosp_barbet-userdebug
     4. aosp_bluejay-userdebug
     5. aosp_bramble-userdebug
     6. aosp_bramble_car-userdebug
     7. aosp_car_arm-userdebug
     8. aosp_car_arm64-userdebug
     9. aosp_car_x86-userdebug
     10. aosp_car_x86_64-userdebug
     11. aosp_cf_arm64_auto-userdebug
     ...
     ...
     ...
     36. aosp_whitefin-userdebug
     37. aosp_x86-eng
     38. aosp_x86_64-eng
     39. arm_krait-eng
Which would you like? [aosp\_arm-eng]
Pick from common choices above (e.g. 13) or specify your own (e.g. aosp\_barbet-eng):
```

After this prompt appears, the developers are free to select the target_variant combination they require. For the generalizability of the builds, during this study the option 'aosp_arm64-eng' is used.

5. Building the AOSP

   After the above lunch process is successful, the developers are shown a prompt which summarizes the Android build to be built and where the final build will be stored. In the output below, the OUT_DIR is shows as 'out', which means the OS will be stored in the folder called out.

```
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=13
TARGET_PRODUCT=aosp_arm64
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_ARCH=arm64
TARGET_ARCH_VARIANT=armv8-a
TARGET_CPU_VARIANT=generic
TARGET_2ND_ARCH=arm
TARGET_2ND_ARCH_VARIANT=armv8-a
TARGET_2ND_CPU_VARIANT=generic
HOST_ARCH=x86_64
HOST_2ND_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-5.10.0-11-amd64-x86_64-Debian-GNU/Linux-11-(bullseye)
HOST_CROSS_OS=windows
HOST_CROSS_ARCH=x86
HOST_CROSS_2ND_ARCH=x86_64
HOST_BUILD_TYPE=release
BUILD_ID=TP1A.221005.002
```

```
  OUT_DIR=out
  PRODUCT_SOONG_NAMESPACES=device/generic/goldfish device/generic/goldfish-opengl
  hardware/google/camera hardware/google/camera/devices/EmulatedCamera
============================================
```

After the above prompt appears, the OS is ready to be built. The final building process might take a few hours depending on the resource availability.

```
1  # Needs to be executed from the root of the AOSP
2  $ m
```

6. Extracting the built OS

   If executed succesfully, the 'out/target' folder within the AOSP root will be populated with the Android OS and will carry the following structure or similar structure depending on the target_variant selected.

```
.
|-common
  |-obj
    |-APPS
    |-JAVA_LIBRARIES
  |-product
    |-generic_arm64
        |-apex
        |-debug_ramdisk
        |-dexpreopt_config
        |-fake_packages
        |-gen
        |-obj
        |-obj_arm
        |-root
        |-system
        |-vendor
```

Chapter 3

Literature Review

This section provides a literature review of prior work related to security and privacy aspects associated with the Android supply chain and vendor customizations.

3.1 Characterizing Android OEM/vendor customizations

Prior studies have analyzed and assessed the effect of the supply chain on security and privacy of the resulting Android devices. The pioneering studies analyzed the Android OS customizations mainly focusing of device drivers (kernel level) and the overall firmware [65, 97, 98, 99]. Another subset of studies focused more specifically on the risks associated to preinstalled applications and vendor customizations. While some focused on critical aspects such as open ports in Android images [96], others characterized OEM customizations, from personal data collection by privileged components [69, 66, 79, 80] to custom permissions [70]. I discuss these seminal papers in more depth next.

Gamba et al. presented the first large-scale study analyzing the pre-installed applications existing across 200 Android vendors, in order to study the user information dissemination through these applications. The authors utilized static analysis, manual analysis and traffic analysis techniques to uncover known malware in system partitions of Android devices, applications requesting dangerous custom permissions and potential dissemination of personally identifiable information (PII). Through the extensive analysis, the authors show that the Android supply chain is a largely complicated ecosystem which is built around the contingencies of user data [69].

Complementing Gamba's work, the work by Liu et al. used traffic analysis methods to report privacy leaks caused by the actual Android OS. Even when the device is at an idle state, the vendor handsets communicate sensitive information such as IMEIs, hardware serial numbers to the respective developers of the Android OS and also to supply chain partners such as social networks (Facebook, LinkedIn) through the relationship made through the preinstalled applications. Another similar study done on comparing the security and privacy of traffic sent by Pixel and iPhone [79] in similar execution environments showed that the even at idle state both Android and iOS based devices share user information and data on average every 4.5 minutes, and this also includes MAC address sharing with nearby devices either IoT or mobile devices. Another study done by Leith revealed how telemetry data (specifically Google

Dialer and Google Messages apps) is sent to Google services such as Firebase analytics. The data sent out such as the hash of message text, call time and durations shows how system apps with privileged permissions and non-opt out behaviors could raise privacy concerns for users.

Compliance related issues inherited due the Android customizations was thoroughly analyzed by Possemato et al. in accordance with the Google's CDD. The authors focused on Android binary customizations (ELF executable and native library layer), mandatory access control (MAC) system or security enhanced linux - SELinux policies, Android init script and kernel security [87]. Out of the 2,907 Android images analyzed, 20% of the Android firmware failed the compliance rules presented in CDD by failing at least one test. Interestingly, this proportion included already Google certified Android OEM/vendor devices which could be used as a indication on the extent of freedom enjoyed by Android developers and the loopholes within the Google compliance test suites.

Finally, Grace et al. revealed different shortcomings in the Android permission system that allows user-installed applications gaining access to unprotected sensitive components exposed by privileged systems applications [72]. A study done on malware detection in Android firmware [98] using static and dynamic analysis showed that preinstalled system applications can act as a distribution channel for malware.[63] analyzed a important dynamic part of the Android supply chain, Firmware over the Air app (FOTA). These applications hold the capacity to updates applications within the system partition as well as install new ones. The authors showed that FOTA apps in the wild display cases of installing malware, potentially unwanted programs and the integration of third party libraries which could result in privacy intrusive behaviors [63]. A recent study on vendor defined Android custom permissions revealed that these permissions are essentially invisible to the end user, therefore contributes to the lack of transparency within vendor supply chains. The researchers used custom tools to detect how custom permissions can act as an enabler for normal apps to access permission-protected Android resources [70].

3.2 Android network security

The rise of TLS/SSL based attacks such as Man-in-the-Middle (MITM) attacks, encouraged the security researchers to pay attention to Android's take on SSL/TLS, both by Android app developers and by device OEMs. A study done on Android SSL vulnerabilities on Android apps show that developer misconfigurations are a leading cause for insecure implementations, as a result of the complex SSL stack. The authors present two package-level solutions for the issue; allowance of any certificate and hostname based on SSL verification debug flag and enabling the SSL pinning in the package manifest. Similarly, MalloDroid [67], confirms these prior findings, pointing at common developer errors such as applications that trust all certificates, allow all host names, mix secure and insecure communication, and trusting too many certificates. Similar studies with similar conclusions, yet using different methodologies, were carried out by Greenwood and Khan and Razaghpanah et al.. The latter shows that most applications tend to gravitate towards using the TLS configurations provided by the OS itself by default. While this could provide a secure implementation rather than a customized TLS/SSL stack this depends on Android OS being up-to-date with the latest security patches and system updates including the TLS libraries,

CA certificates. Yet, some developers implement their own TLS libraries to protect their applications from vulnerabilities introduced at the operating system [89].

Pradeep et al. conducted an analysis on one of the most popular security mechanisms used by application developers: certificate pinning. The authors discuss the importance of providing developers with proper programmatic support to facilitate certificate pinning functions. Otherwise, developers are compelled to use their own TLS implementations or rely on third-party TLS libraries, which have been proven to be inconsistent and introduce various threats into the Android ecosystem [88].

In fact, developer practices and their shortcomings play a significant role in the Android supply chain, a study done on the developer's dependency on online programming discussion platforms like stackoverflow on Java security libraries including JCA, JCE and JSSE [68]. The application level copy and pasted insecure code that is handling TLS connections was 14% of the 200K application the authors analyzed, where a significant portion of copied code was detected in javax.crypto, javax.net.ssl and java.security.

Yet, these prior studies focus on Android app developer practices, outside the scope of this thesis. A very specific study done on low-level Android OpenSSL's pseudo-random number generator by Kim et al. discovered that Android security could be threated by giving the ability partially recovering the PreMasterSecret of the initialized SSL sessions [75]. Blessing et al. conducted an extensive study on all the open source cryptographic libraries which has the potential to server as the crypto provider in Android OS. The authors focus on C/C++ based libraries due to the potential of not memory safe unlike java, the findings showed that the public disclosure of vulnerabilities in crypto libraries are far less that non-crypto libraries. The authors show that Google has kept BoringSSL relatively secure and small in size by removing 70% of the original OpenSSL codebase (but at the time of this study the size of BoringSSL has being growing at an accelerating phase) and in default discards many of the outdated ciphers [64]. Due to BoringSSL being relatively secure, the authors conclude this as a safe alternative for OpenSSL in Android but any changes done to the network stack might overshadow this secureness by introducing security implications.

A TLS layer security analysis based on the proper deletion of cryptographic keys once they have served their purpose, shows the complicated interface between C and Java code that promotes the potential for memory disclosure attacks in Android [78]. The authors conclude that the session cache deletion process in Conscrypt is problematic in the current Android TLS/SSL stack.

A recent study looking at the large scope of vulnerabilities in Android, security patches and updates [74] revealed how Android images from large vendors such as Realme issue vulnerable Android images 90% of the time. The authors highlight that almost all the Android vendors focus on feature updates rather than applying critical security patches,which results in a device and in relationship the user being vulnerable to external threats for as much as 3 months.

Finally, Vallina-Rodriguez et al. gave insight into the supply chain's effect on the root store certificates by showing that a considerable amount of certificate are added on top of the default root store from the AOSP by MNOs, government agencies and point out that the certificate root store is a strong decision point in Android SSL/TLS security [93]. Though not limited to Android, Ma et al. presents the first large scale root

store ecosystem exploration. The authors shows that Android device vendors could take as long as 430 days to remove high severity CA certificates from their devices, which shows the importance of proper sanitation of Android network stack [82].

Chapter 4

Research Methodology

This chapter describes the methodology followed to achieve the dissertation objectives listed in the Introduction chapter. This chapter is structured as follows: Section 4.1 describes the data sources and the process followed for gathering Android firmware images at scale. Section 4.2 describes the process for extracting the network stack packages and libraries. I note that this step requires different methods of decompilation based on the Android version and thevendors. After the successful extraction of JSSE , JCE and JCA critical components, Section 4.3 presents the static analysis methods that I implemented to (i) detect vendors' customizations, and (ii) evaluate the potential introduction of security vulnerabilities and threats. The whole methodology is visualized in Figure 4.1, describing the different steps that compose my analysis pipeline.

4.1 Data Collection

In order to conduct our empirical and exhaustive analysis of vendor customizations, it is important to have access to a diverse set of Android images, from different vendors and Android versions. To that end, We rely on two complementary datasets compiled by the research and cybersecurity community: Firmware Scanner [36] and Android dumps [60]. FirmwareScanner gathers Android firmware files through crowdsourcing while Android dumps and official images are manually downloaded from third-party and vendor websites. The manually download firmware images are used to validate the credibility of the firmware files collected through crowd-sourcing and also to extract network related components from some vendor devices which were lacking from the FirmwareScanner dataset but were necessary to the analysis. In the next subsections, I describe each data source in detail.

4.1.1 Firmware Scanner

The primary method of data collection depends on the FirmwareScanner app [36], which is developed and maintained by IMDEA Networks Institute's internet Analytics Group (IAG). This tool is publicly available in Google Play Store and serves the purpose of crowdsourcing preinstalled applications, certificates and other executables, binaries using crowd-sourcing means; as for the OS partitions Firmware Scanner is able to scan system, vendor, ODMs and product partitions.

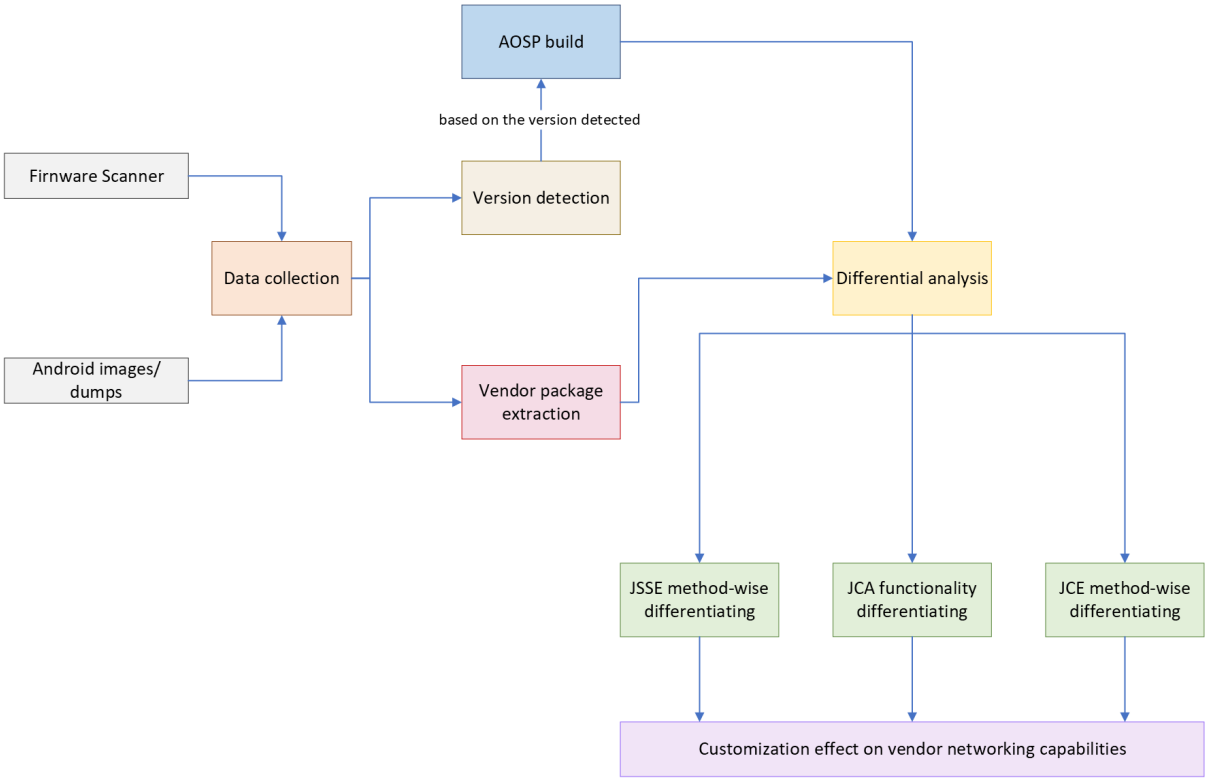


Figure 4.1: Overview of the methodology followed in order to detect vendor customization effect on secure communication.

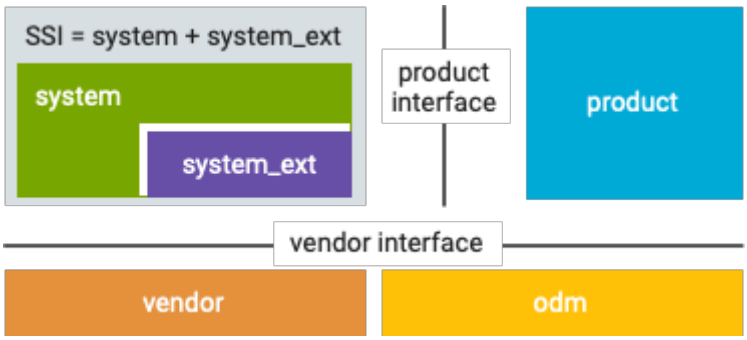


Figure 4.2: System partitions scanned by Firmware Scanner (Source : `https://source.android.com/docs/core/architecture/partitions`).

The system partition is critical for the network stack analysis since this contains the Android framework, which is the parent folder for native libraries, apex files and core system modules. ODM contains the original design manufacturer (ODM) customizations done on the HAL and the kernel modules, while the vendor partition contains the any binary files that doesn't belong in the AOSP stack which are the proprietary packages or libraries. Finally, product partition is used by Android developers or vendors to install product-specific module, which can also be interpreted as an extension on the system partition. This includes files such as build.prop, product specific native libraries, java libraries and apps.

Once installed in a user device, Firmware Scanner requests user consent to start

the scanning of the device. If the user consent is granted the tools will start with computing hashes to perform a server-side checksum comparison to see if the file already exists in the database, if the files do not exists then they are queued to be uploaded to our servers. Along with the files that have been scanned, some meta data regarding the device will also be recorded; such as the Android build fingerprint, Android version the device is running, device model,name and manufacturer and the timezone of the device. Once the firmware scanner send the files to the server and finishes the analysis, it shows the Android device user the number of files and the types of files detected (Figure 4.3 shows the steps the user see while using the FirmwareScanner).

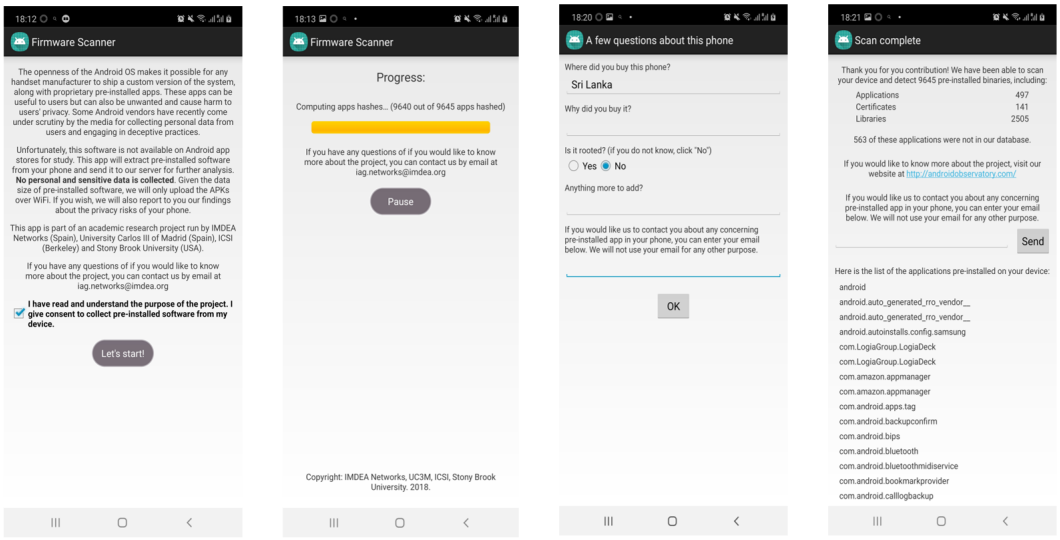


Figure 4.3: FirmwareScanner's UI screenshots showing the data collection.

4.1.2 Android Dumps

While Firmware Scanner is used as the primary data source for this analysis, its coverage for newer Android versions is limited. To overcome this limitation, we rely on a secondary data source; Android dumps [60] which gives us access to official firmware versions from vendors (OEMs) like AT&T, Blackshark, Hisense, LGE, Oneplus, Oppo, Redmi, TCL, Tecno, Huawei, Asus, ZTE, Nokia, Motorola and Alps, since Android 9.

Like FirmwareScanner, Android Dumps relies on crowd-sourcing approaches. However, it is a publicly available and free web platform where users from around the world can upload their Android firmware images into a common repository. These images are sometimes saved as .zip files or .img files, therefore suitable methods of file extraction needs to be followed in order to extract their TLS/SSL implementation. The overview of the Android firmware images collected for this research is shown in Table 4.1.

Versions	Number of devices collected	Number of vendors
9	18005	193
10	16433	176
11	10719	115
12	2952	47
13	411	13

Table 4.1: Summary of the data collected through the FirmwareScanner and Android Dumps.

Since the focus of the study is studying the core components of the network stack in Android devices, it is important to explore the location of these classes and libraries to extract the maximum information from an Android device. Table 4.2 presents the default locations and the files and libraries which could be found in each of the locations that will be used in this study.

File type	Default location	Potential networking components
boot.oat	/system/framework/arm/boot.oat /system/framework/arm64/boot.oat	Includes the core-oj components including javax.net, javax.crypto etc which are sourced by OpenJDK
core-oj.jar	/system/framework/core-oj.jar */apex/com.android.art.debug/javalib/core-oj.jar	Includes the core-oj components including javax.net, javax.crypto etc which are sourced by OpenJDK
Root certificates	/etc/security/cacerts	Includes the root store certificates of each device
Native libraries	*/lib */lib64	Includes the compiled libraries built during runtime including the JCA providers
Conscrypt	/framework/arm/boot-conscrypt.oat /framework/arm64/boot-conscrypt.oat /framework/conscrypt.odex /framework/conscrypt.jar */apex/com.android.conscrypt/javalib/conscrypt.jar	Includes the source code for conscrypt, could be either in .oat, .jar or .odex formats
Okhttp	/framework/arm/boot-okhttp.oat /framework/arm64/boot-okhttp.oat /framework/okhttp.odex /framework/okhttp.jar */apex/com.android.art.debug/javalib/okhttp.jar	Includes the source code for okhttp, could be either in .oat, .jar or .odex formats
Bouncy Castle	/framework/arm/boot-bouncycastle.oat /framework/arm64/boot-bouncycastle.oat /framework/bouncycastle.odex /framework/bouncycastle.jar */apex/com.android.art.debug/javalib/bouncycastle.jar	Includes the source code for bouncy castle, could be either in .oat, .jar or .odex formats

Table 4.2: The default locations and probable locations for the networking libraries, packages and core components.

The default location of these files could change across vendors. As a result, the images must be carefully explored to find the correct location of each file. The `apex/` (Android pony express) folder and structure was introduced to the Android OS in Android version 10 as way to deliver low-level system modules in an Android device. While the build of an Android device is straight-forward the reversing of different files requires different techniques, each of these approaches and the tools used will be discussed in Section 4.2.

4.2 Extracting SSL/TLS packages

4.2.1 Java Secure Socket Extension (JSSE) packages

The JSSE packages in Android exists in the `core-oj.jar` or in the `boot.oat` files. While the source of the JSSE, OpenJDK is common to each vendor depending on the build version and the developer specific Android architecture, the destination of the compiled code might vary. The files could exists in either `boot.oat` and `core-oj.jar` as shown in table 4.2.

The initial `core-oj/` structure according to the AOSP is as follows in smali code:

```
.core-oj
    |-com
        |--sun
    |-java
        |--awt
        |--beans
        |--io
        |--lang
        |--math
        |--net
        |--nio
        |--security
        |--sql
        |--text
        |--time
        |--util
    |-javax
        |--annotation
        |--crypto
        |--net
        |--security
        |--sql
    |-jdk
        |--internal
        |--net
    |-sun
        |--invoke
        |--misc
        |--net
        |--io
        |--reflect
        |--security
        |--util
```

Decompiling the boot.oat. The important classes that make up JSSE API are within `javax/net` folder, shown in Table 2.1. In order to reverse an oat file, it's important to understand the underlying process of creating an `.oat` file from their source `.java` files. The overview of the complete process in shown in Figure 4.4.

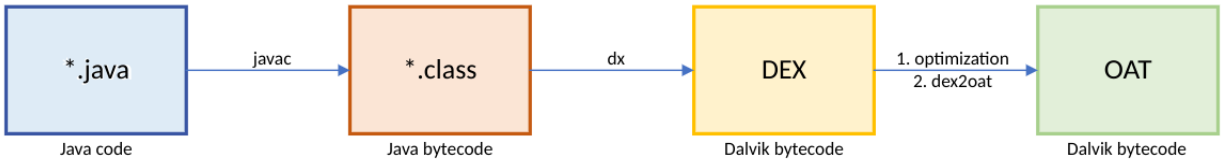


Figure 4.4: Java to oat compilation for Android (applies to all the `.oat` files found within an Android device).

The dex to oat conversion includes two subprocesses. The first step involves applying optimization techniques to decrease package size and improve performance using tools like ProGuard (or DexGuard). ProGuard [41] is a tool to optimize code that removes any unused fields, methods or classes and then applies obfuscation, thus making it hard for humans to reverse-engineer and understand the binary object. DexGuard, although it is similar to ProGuard, it is specifically targeted for Android OS code. After the optimization step the resulting java bytecode is converted to its architecture specific Dalvik bytecode. Therefore, reverse-engineering requires the execution of unpacking and disassembly applied from oat to Java. For a `boot.oat` file, the following steps produce the Java representation of the initial source code.

```
1   #oat TO Java
2   #Step 1: use oat2dex to convert .oat to .dex (This will deodex the boot file)
3   java -jar oat2dex.jar -o <output path> boot <boot.oat path>
4
5   #Step 2: use dex2jar to convert the resulting .dex files in to their respected .jar
        representations which encapsulates the java bytecode.
6   ./d2j-dex2jar.sh <dex_input path> -o <output path>
7
8   #Step 3: use fernflower decompiled to convert the .jar file from bytecode to java code and
        decompress the resulting .jar file/
9   java -jar fernflower.jar <input_bytecode_jar path> -o <output_java_code_jar path>
10  unzip <output_java_code_jar path> -d <output_java_code path>
```

Listing 4.1: Reversing `.oat` to `.java`.

I use existing tools such as baksmali git [31], dex2jar git [30] and fernflower [28] for reversing, all of which are open source tools widely used by the Android research community [69, 71, 76]. As shown in the Java to oat compilation process, deobfuscation through optimization techniques are applied to high level languages such as Java. However, these techniques do not directly impact on low-level machine code such as smali code. In simple terms, smali for Android Java code is equivalent to assembly for C language; therefore in order to have a deobfuscation tolerant representation of the network components the oat files are also disassembled to create the smali representation as well.

```
1   #oat to smali
2   #Step 1: use baksmali to disassemble the boot.oat file
3   java -jar baksmali.jar dis <boot.oat path> -d <framework_folder path> -o <output path>
```

Listing 4.2: baksmali usage for decompiling `.oat` files.

For this step, I use baksmali [29]. Even though the smali representation is not human readable, it presents a proper overview of the class structure. From example following

is the smali representation of the javax.net.SocketFactory class.

```
.class public abstract Ljavax/net/SocketFactory;
.super Ljava/lang/Object;
.source "SocketFactory.java"


# static fields
.field private static theFactory:Ljavax/net/SocketFactory;


# direct methods
.method protected constructor <init>()V
    .registers 1

    .prologue
    .line 82
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    return-void
.end method

.method public static getDefault()Ljavax/net/SocketFactory;
    .registers 2

    .prologue
    .line 92
    const-class v1, Ljavax/net/SocketFactory;

    monitor-enter v1

    .line 93
    :try_start_3
    sget-object v0, Ljavax/net/SocketFactory;->theFactory:Ljavax/net/SocketFactory;

    if-nez v0, :cond_e

    .line 100
    new-instance v0, Ljavax/net/DefaultSocketFactory;

    invoke-direct {v0}, Ljavax/net/DefaultSocketFactory;-><init>()V

    sput-object v0, Ljavax/net/SocketFactory;->theFactory:Ljavax/net/SocketFactory;
    :try_end_e
    .catchall {:try_start_3 .. :try_end_e} :catchall_12

    :cond_e
    monitor-exit v1

    .line 104
    sget-object v0, Ljavax/net/SocketFactory;->theFactory:Ljavax/net/SocketFactory;

    return-object v0
```

Listing 4.3: smali representation of javax.net.SocketFactory.

As we can observe in this code snippet, we can observe how the class `Ljavax/net/SocketFactory` includes an initialization method and a getDefault method which returns a `Ljavax/net/SocketFac` type object.

Decompiling the core-oj.jar, `core-oj.jar` which is a Java archive can be decompiled using baksmali.

```
    #.jar to smali
    #Step 1: use baksmali to convert .jar to smali
    java -jar baksmali.jar dis <core-oj.jar path> -o <output path>
```

Listing 4.4: baksmali usage for decompiling `.oat`.

4.2.2 Java Cryptographic Architecture (JCA) providers

In Android JCA provider by default (or as implemented in the AOSP) is BoringSSL. BoringSSL is implemented using C/C++, and it compiles into two .so files: `libssl` and `libcrypto`.These are stored in `lib/` and `lib64/` folders. Since these are object files, they can only be analyzed through assembly instructions and by studying their external dependencies. Due to them exisiting as shared objects, several methods of object analysis techniques are considered.

Objdump [45] is a popular linux tool used to extract and display information from object files. Using objdump, the external dependencies of a `.so` file can be detected. For example the following `libssl.so` file shows that it depends on `libcrypto.so`. libcrypto object file consists the cryptographic algorithms and operations that is used by `libssl.so` to implement secure communication protocols. These dependencies can indicate any additional cryptographic primitive providers that vendors might be using.

```
1 $ objdump -x system/lib64/libssl.so | grep 'NEEDED'
2   NEEDED               libcrypto.so
3   NEEDED               libc.so
4   NEEDED               libm.so
5   NEEDED               libdl.so
```

Listing 4.5: Dependencies between shared objects.

The .so file can be represented using its' symbol table which holds information needed to locate the program's symbolic definitions which I used to extract function names and function calls between libssl and libcrypto. The symbol tree for a given object file can be retrieved using multiple tools such as the objdump, elf and nm.

nm [44] provides a clear overview of the symbol table and can be used to filter out global functions and global text. The linux nm command displays symbol information from an object file or a library by providing a list of symbols which are defined, undefined or referenced within the file.

```
1 $ nm -D system/lib64/libssl.so
2                  U abort@LIBC
3                  U BIO_callback_ctrl
4                  U BIO_clear_retry_flags
5                  U BIO_ctrl
6                  U BIO_find_type
7                  U BIO_flush
8                  U BIO_free
9 000000000001b014 T BIO_f_ssl
```

Listing 4.6: Symbol table for libssl.so.

The above listing includes all the dynamic symbols including defined and undefined symbols, which means this could include symbols which are defined internally to the shared object and any symbols which are external (but used within the object). To distinguish between these two cases, the nm command is modified as described

below. The addition of the –defined-only flag provides the symbols which have a defined implementation within the libssl object file.

```
1  $ nm -D --defined-only system/lib64/libssl.so
2  00000000000121fc T BIO_f_ssl
3  0000000000012208 T BIO_set_ssl
4  00000000000600e0 A __bss_start
5  000000000002e5a4 T d2i_SSL_SESSION
6  000000000002e528 T d2i_SSL_SESSION_bio
7  ...
8  ...
9  ...
10 000000000002bc08 T _Z23SSL_CTX_sess_set_get_cbP10ssl_ctx_stPFP14ssl_session_stP6ssl_stPhiPiE
11 000000000003c2a8 T _ZN4bssl10OpenRecordEP6ssl_stPNS_4SpanIhEEPmPhS3_
12 000000000003c54c T _ZN4bssl10SealRecordEP6ssl_stNS_4SpanIhEES3_S3_NS2_IKhEE
13 00000000002530c T _ZN4bssl14CBBFinishArrayEP6cbb_stPNS_5ArrayIhEE
14 000000000002a194 T _ZN4bssl15SSL_SESSION_dupEP14ssl_session_sti
15 00000000000155dc T _ZN4bssl17SSL_apply_handoffEP6ssl_stNS_4SpanIKhEE
16 0000000000015a40 T _ZN4bssl18SSL_apply_handbackEP6ssl_stNS_4SpanIKhEE
17 000000000003c3ac T _ZN4bssl19SealRecordPrefixLenEPK6ssl_stm
18 000000000003c434 T _ZN4bssl19SealRecordSuffixLenEPK6ssl_stm
19 0000000000015594 T _ZN4bssl19SSL_decline_handoffEP6ssl_st
20 00000000000154b0 T _ZN4bssl21SSL_serialize_handoffEPK6ssl_stP6cbb_st
21 0000000000015760 T _ZN4bssl22SSL_serialize_handbackEPK6ssl_stP6cbb_st
22 0000000000025a78 T _ZN4bssl24SSL_CTX_set_handoff_modeEP10ssl_ctx_stb
```

Listing 4.7: Defined only symbols in shared object.

The middle column of the output indicates the type of symbol while the first column shows the symbol value. For symbol type 'T' indicates that the respective symbol is defined exported symbols while 'A' indicates an defined absolute value. Looking at the example of defined export symbols, in Listing 4.7 we can see how it includes low-level symbols that are not human understandable from line 10 onwards. This requires converting these into user-level names, which outputs the human-readable C/C++ function names.

```
1  $ nm -D --defined-only --demangle system/lib64/libssl.so
2  00000000000121fc T BIO_f_ssl
3  0000000000012208 T BIO_set_ssl
4  00000000000600e0 A __bss_start
5  000000000002e5a4 T d2i_SSL_SESSION
6  000000000002e528 T d2i_SSL_SESSION_bio
7  ...
8  ...
9  ...
10 000000000002bc08 T SSL_CTX_sess_set_get_cb(ssl_ctx_st*, ssl_session_st* (*)(ssl_st*,
       unsigned char*, int, int*))
11 000000000003c2a8 T bssl::OpenRecord(ssl_st*, bssl::Span<unsigned char>*, unsigned long*,
       unsigned char*, bssl::Span<unsigned char>)
12 000000000003c54c T bssl::SealRecord(ssl_st*, bssl::Span<unsigned char>, bssl::Span<unsigned
       char>, bssl::Span<unsigned char>, bssl::Span<unsigned char const>)
13 00000000002530c T bssl::CBBFinishArray(cbb_st*, bssl::Array<unsigned char>*)
14 000000000002a194 T bssl::SSL_SESSION_dup(ssl_session_st*, int)
15 00000000000155dc T bssl::SSL_apply_handoff(ssl_st*, bssl::Span<unsigned char const>)
16 0000000000015a40 T bssl::SSL_apply_handback(ssl_st*, bssl::Span<unsigned char const>)
17 000000000003c3ac T bssl::SealRecordPrefixLen(ssl_st const*, unsigned long)
18 000000000003c434 T bssl::SealRecordSuffixLen(ssl_st const*, unsigned long)
19 0000000000015594 T bssl::SSL_decline_handoff(ssl_st*)
20 00000000000154b0 T bssl::SSL_serialize_handoff(ssl_st const*, cbb_st*)
21 0000000000015760 T bssl::SSL_serialize_handback(ssl_st const*, cbb_st*)
22 0000000000025a78 T bssl::SSL_CTX_set_handoff_mode(ssl_ctx_st*, bool)
```

Listing 4.8: Defined only symbols in shared object converted to user-level.

The example below shows how to filter undefined but used symbols within the shared object, the output values could indicate a dependency between two shared objects.

```
$ nm -D --undefined-only system/lib64/libssl.so
U abort
U ASN1_d2i_bio
U ASN1_i2d_bio
U BIO_callback_ctrl
U BIO_clear_retry_flags
U BIO_copy_next_retry
U BIO_ctrl
U BIO_find_type
U BIO_flush
U BIO_free
U BIO_free_all
U BIO_get_fd
U BIO_get_retry_reason
U BIO_method_type
U BIO_new
U BIO_read
U BIO_read_filename
```

Listing 4.9: Undefined only symbols in shared object.

As shown in the `libssl.so` dependencies (listing 4.5), the shared object libssl depends on libcrypto share object. This relationship can be systematically visualized using the command explained above.

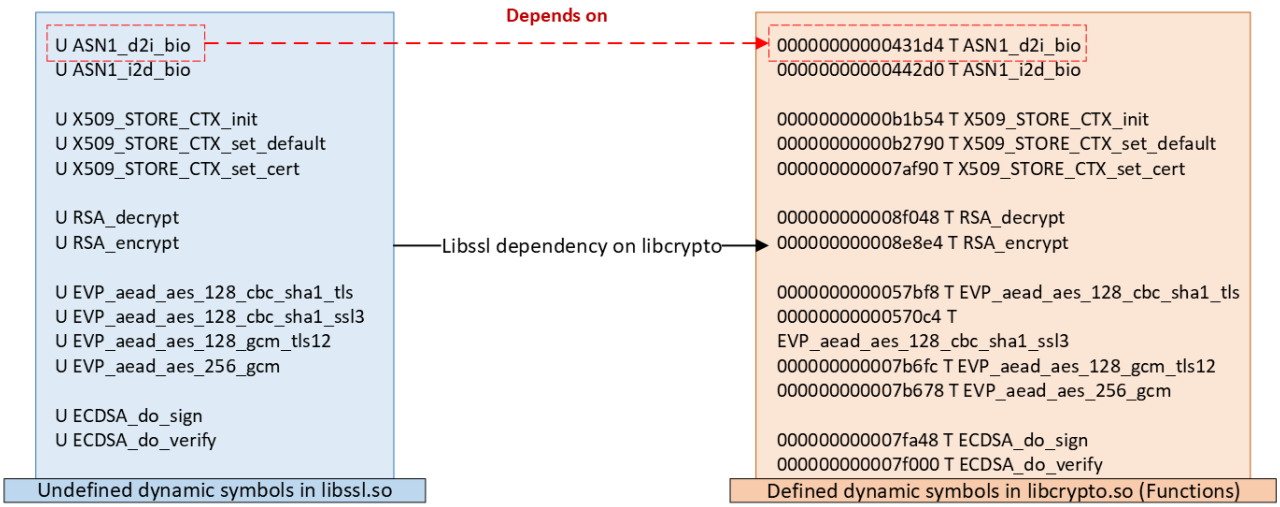


Figure 4.5: BoringSSL internal dependencies, between libssl and libcrypto shared objects.

The source code files of shared objects provides insight into the cryptographic provider source code folder within the vendor Android architecture. For this task, I used the **strings** command ; for example if the C/C++ source files are compiled into the .so following command can be used to see which sources were exactly compiled. This indicates that the `libssl.so` under examination is compiled using boringssl.

```
$ strings -a system/lib64/libssl.so | grep '\.c'
external/boringssl/src/ssl/custom_extensions.cc
external/boringssl/src/ssl/d1_both.cc
```

```
external/boringssl/src/ssl/d1_lib.cc
external/boringssl/src/ssl/d1_pkt.cc
external/boringssl/src/ssl/d1_srtp.cc
external/boringssl/src/ssl/dtls_method.cc
external/boringssl/src/ssl/dtls_record.cc
external/boringssl/src/ssl/handshake.cc
external/boringssl/src/ssl/handshake_client.cc
external/boringssl/src/ssl/handshake_server.cc
external/boringssl/src/ssl/s3_both.cc
external/boringssl/src/ssl/s3_pkt.cc
external/boringssl/src/ssl/ssl_aead_ctx.cc
external/boringssl/src/ssl/ssl_asn1.cc
external/boringssl/src/ssl/ssl_buffer.cc
external/boringssl/src/ssl/ssl_cert.cc
external/boringssl/src/ssl/ssl_cipher.cc
external/boringssl/src/ssl/ssl_file.cc
external/boringssl/src/ssl/ssl_key_share.cc
external/boringssl/src/ssl/ssl_lib.cc
external/boringssl/src/ssl/ssl_privkey.cc
```

Listing 4.10: 'Source code files compiled into libssl.so.

4.2.3 Java Cryptographic Extension (JCE) packages

The current Android network stack includes two Java Cryptographic Extension (JCE) packages: Conscrypt and BouncyCastle. Although the developers are free to incorporate any custom or third-party package that facilitates the purpose of the JCE, these two packages are the current standard practice within the AOSP with Conscrypt as the default JCE provider (Section 2.3.1.3).

Decompiling Conscrypt. Conscrypt can be decompiled using the above mentioned techniques, depending on the file type. For `.oat` and `.jar` files the methods described in listings 4.2 and 4.4 are used, respectively. There could be another instance where conscrypt could be encapsulated in a `.odex` file. In this case, I use baksmali.

```
#.jar to smali
#Step 1: use baksmali to convert .odex to smali
java -jar baksmali.jar dis <conscrypt.odex> -o <output path>
```

Listing 4.11: baksmali usage for decompiling `.odex` files.

After the decompilation the conscrypt packages will include the structure as follows,

```
.conscrypt
  |-com
    |-android
      |-org
        |-conscrypt
          |-ct
```

Decompiling BouncyCastle. Similarly to conscrypt, `.oat`, `.jar`, and `.odex` files can be decompiled using baksmali. The decompiled Bouncy castle includes a complicated structure compared to other packages included in the Android network stack.

```
.bouncycastle
  |-android
    |-org
      |-bouncycastle
        |-asn1
          |-bc
```

```
        |-cms
        |-eac
        |-iana
        |-isismtt
        |-kisa
        |-misc
        |-nist
        |-ntt
        |-oiw
        |-pkcs
        |-sec
        |-teletrust
        |-util
        |-x500
          |-style
        |-x509
        |-x9
      |-crypto
        |-agreement
        |-digests
        |-ec
        |-encodings
        |-engines
        |-generators
        |-io
        |-macs
        |-modes
          |-gcm
        |-paddings
        |-params
        |-signers
        |-util
      |-jcajce
        |-provider
          |-asymmetric
          |-dh
          |-dsa
          |-ec
          |-rsa
          |-util
          |-x509
        |-config
        |-digest
        |-keystore
          |-bc
          |-pkcs12
        |-symmetric
          |-util
        |-util
      |-spec
      |-util
    |-jce
      |-exception
      |-interfaces
      |-netscape
      |-provider
      |-spec
    |-math
      |-ec
        |-custom
          |-sec
        |-endo
      |-field
      |-raw
    |-util
      |-encoders
      |-io
        |-pem
    |-x509
      |-extension
```

4.2.4 Android HTTPS providers

OkHttp can be decompiled similar to Conscrypt using baksmali according to each file type. The overview of the structure of OkHttp is as follows:

```
.Okhttp
  |-com
    |-android
      |-Okhttp
        |-internal
          |-tls
          |-http
          |-io
          |-framed
```

```
|-okio
```

4.3 Detecting Vendor Customizations

By exploring each device's Java and C/C++ networking packages , we study vendor customizations and their effect on TLS/SSL security. Yet, this analysis requires a meticulous approach. Our methodology involves the following steps:

1. Establishing the baseline for the differential analysis between vendors.
2. Implemeting a generalizable framework for differential analysis
3. Applying the differential techniques (i.e., diffing) to each of the SSL/TLS stack core components - Characterzing the vendor customizations on the Android TLS/SSL stack
   (a) Extracting the vendor customizations on the JSSE API layer
   (b) Extracting the vendor customizations on the JCE layer
   (c) Extracting the vendor customizations on the native library JCA providers

In order to properly detect and extract vendor customizations done by vendors on a AOSP derivative, we set a proper baseline. This will act as the ground truth when comparing the OEM/vendor manufactured Android devices, and provides an opportunity to compare and delve into (i) vendor specific modifications, and (ii) their purpose. The process of establishing the baseline is described in Section 4.3.1, and how the baseline will be utilized in analyzing the vendor's adaptation of JSSE, JCA packages and JCA providers is presented in Sections 4.3.2, 4.3.4 and 4.3.3, respectively.

4.3.1 Establishing the Baseline

The ground truth selection within this study determines the success of properly defining the customizations done by the vendors. A careful examination of the past literature showed that, while some developers used the original AOSP source [1] as a reference point [87], some researchers have opted for Google Pixel or Nexus devices as a reference point [74].

The varying versions and builds collected through in the firmware dataset being explored in the study, requires the availability of each Android build version (either AOSP sourced build or Google Android devices) in order to produce a stable differe analysis of each vendor. This acts as an important indicator when selecting the baseline between the AOSP build or Google manufactured Android handsets. Collection of Pixel/Nexus firmware for each android version build could be a daunting task given the plethora of devices that needs to be scanned to narrow down the wanted Google device firmware. But due to openess and the availability of the versioning information of the AOSP source code, it is possible to build each Android version required for the analysis. Therefore AOSP builds for each Android version/build will be used as the baseline models in this study. The techniques used to build the AOSP builds from the source are described section 2.4.

[1] It will be referred to as the AOSP build in the study.

After establishing the baseline for the study, the technique of differentiating between vendor's AOSP derivative and the AOSP build should be established. As mentioned in Section 4.2, the Java code in the Android source goes through multiple operations in order to become machine executable code that could be executed on Android hardware.

When selecting a direct comparison model for characterzing the vendor customizations, a fair representation of the OEM's Android OS and Google's Android OS should be established. Since the Java code goes through optimizations and obfuscations during the compilation process the disassembled Java code might present some differential results due to the optimization techniques applied that aren't necessarily vendor modifications. For example the below code snippet in listings 4.12 and 4.13 shows the reversed Java code for the `javax.net.SSLSocketFactory` class for the original AOSP source and vendor implementations respectively.

```
1 public abstract class SSLSocketFactory extends SocketFactory
2 {
3     private static int lastVersion = -1;
4     static final boolean DEBUG;
5     static {
6         String s = java.security.AccessController.doPrivileged(
7             new GetPropertyAction("javax.net.debug", "")).toLowerCase(
8                                                             Locale.ENGLISH);
9         DEBUG = s.contains("all") || s.contains("ssl");
10    }
11    private static void log(String msg) {
12        if (DEBUG) {
13            System.out.println(msg);
14        }
15    }
16 ....
17 }
```

Listing 4.12: AOSP source code.

```
1   public abstract class SSLSocketFactory extends SocketFactory {
2     static final boolean DEBUG;
3     private static SSLSocketFactory defaultSocketFactory;
4     private static int lastVersion = -1;
5
6     static {
7        String var0 = ((String)AccessController.doPrivileged((PrivilegedAction)(new
             GetPropertyAction("javax.net.debug", "")))).toLowerCase(Locale.ENGLISH);
8        boolean var1;
9        if (!var0.contains("all") && !var0.contains("ssl")) {
10          var1 = false;
11       } else {
12          var1 = true;
13       }
14
15       DEBUG = var1;
16    }
17
18    private static void log(String var0) {
19       if (DEBUG) {
20          System.out.println(var0);
21       }
22
23    }
24
25  ....
26  }
```

Listing 4.13: Vendor source code.

Although the logic behind the code snippets above are the same, the Java code visually provides a different textual representations. Therefore, in order to perform a fair comparison and a model, an alternative needs to be defined or chosen.

The unaffectability of the optimization and obfuscation techniques on the smali code (as explained in Section 4.2), promotes itself over the high-level Java code as the fair representation for the Vendor vs. AOSP differentiating model.

4.3.2 Java Secure Socket Extension (JSSE) packages

The extraction of the JSSE packages is as mentioned in section 4.2. After the identification of the JSSE package location within the OEM firmware files, each respective smali class file is scanned in order to explore the vendor modifications. In order to characterize the modifications, two enumeration and differential analysis techniques are carried out.

1. Calculate the edit distance between AOSP and vendor javax classes which are used when establishing secure communication.
2. Create a differential technique to detect the method additions and removals carried on by vendors.

In order to produce and overview of the vendor customizations and enumerate the changes done on the JSSE package, edit distance between the AOSP default JSSE classes and the vendor JSSE classes are calculated. The techniques used for calculating the edit distance is explained in section 4.3.2.1. After, the extent of the vendor customizations are summarized and calculated using edit distance. The exact modifications are examined in Section 4.3.2.1.

4.3.2.1 Edit Distance for JSSE packages

The edit distance for smali code can be calculated using hashing techniques such as SHA256, MD5, SHA1 and also advanced hashing methods such as fuzzy hashing. For performance reasons, I rely on the latter.

Fuzzy hashing is a popular diffing technique within the security research community. It is an effective and scalable method for computing the similarities between almost identical files using piece-wise hashing (CTPH) techniques. For example, one of the most popular and trusted fuzzy hashing techniques used for Android malware analysis is **ssdeep**. Ssdeep [59] converts the file into distinct segments using a rolling hash and then generates a 6 bit value for each segmentm. Af a final stage, it concatenates the generated hash values into a final hash digest.

The ssdeep hash of a file can be calculated by inputing the smali code as a string to the hash function of ssdeep.

```
1  import ssdeep
2
3  file = (<file to hash>, encoding='utf-8')
4  content = file.read()
5  hash = ssdeep.hash(content)
```

Listing 4.14: Calculating the hash using ssdeep.

After calcuating the fuzzy hashes using ssdeep, the similarity between two hashes can be calculated using the compare function provided by ssdeep.

```
import ssdeep

file_1 = (<file_1 to hash>, encoding='utf-8')
content_1 = file_1.read()
hash_1 = ssdeep.hash(content_1)

file_2 = (<file_2 to hash>, encoding='utf-8')
content_2 = file_2.read()
hash_2 = ssdeep.hash(content_2)

similarity = ssdeep.compare(hash_1 , hash_2)
```

Listing 4.15: Calculating file similarity using ssdeep.

The resulting similarity will be as a percentage from 0 to 100. If the similarity is 0 that means there are no similarility and a 100 indicates that the two files are similar or the dissimilarities found are negligable. When applying ssdeep to calcualte the changes in vendor devices, the methodology followed is as shown the Figure 4.6.

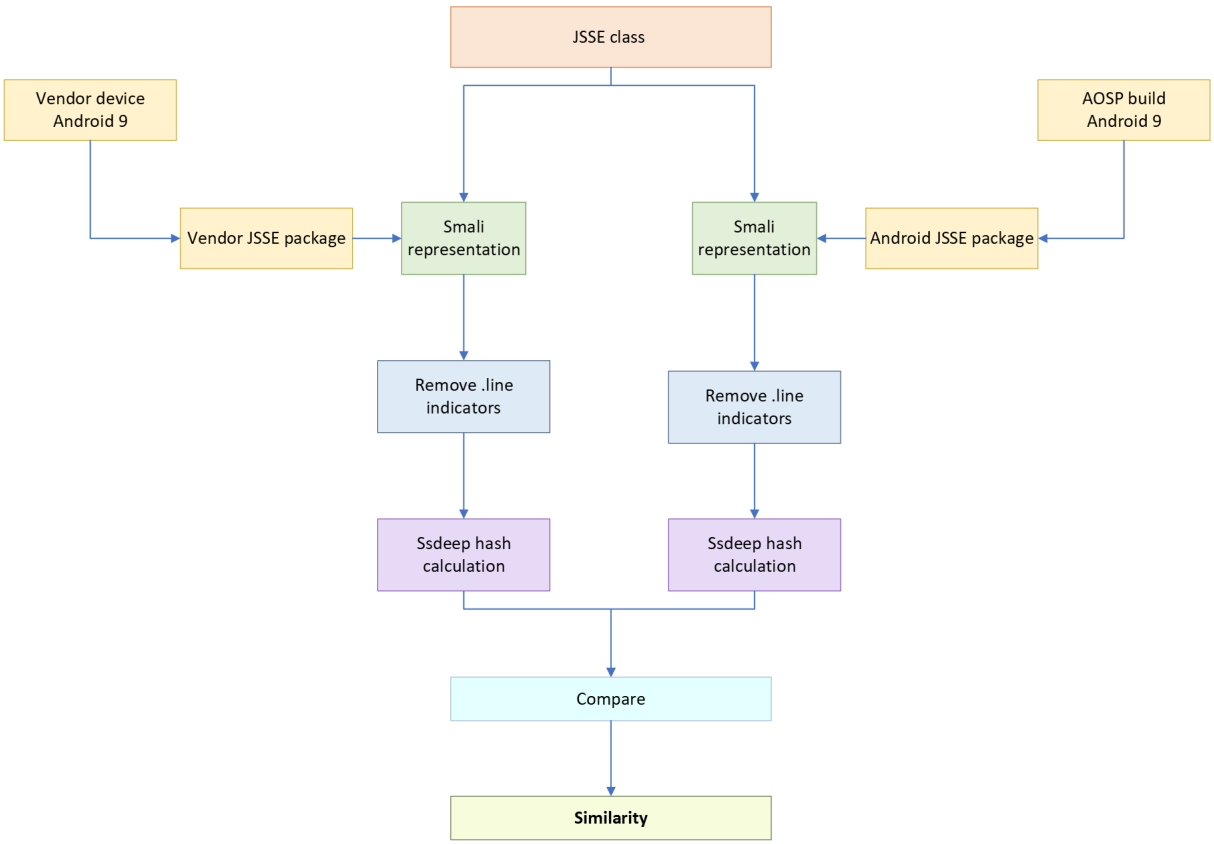


Figure 4.6: JSSE class hash calculation and similarility comparison.

The smali representation of each Java class includes entries starting with the anno-

tation ".line". Simply the number that follows .line indicates which line of code with reference to the original Java code is represented using smali, immediately following the shown line number.

```
1   .method public static getDefault()Ljavax/net/SocketFactory;
2     .registers 2
3
4     .prologue
5     .line 92
6     const-class v1, Ljavax/net/SocketFactory;
7
8     monitor-enter v1
9
10    .line 93
11    :try_start_3
12    sget-object v0, Ljavax/net/SocketFactory;->theFactory:Ljavax/net/SocketFactory;
13
14    if-nez v0, :cond_e
```

Listing 4.16: smali representation of the Java method with line numbers.

The above code snippet shows that the smali representation includes Java code of the method getDefault() within the class `javax.net.SocketFactory`. Specifically, the code shown on lines 92 and 93. However, these line number could vary between vendors and the AOSP code due to vendor modifications such as added/removed lines of code, added/removed methods, or added/removed comments. Since these line numbers could be interpreted as a noticable difference when using CTPH techniques, during the hash calculation these are filtered in order to perform a more accurate detection of changes.

After filtering out line indicators from the smali code, each class from vendor and also the AOSP build are hashed using ssdeep. Then using the hash.compare() function the two hashes can be compared. Example outputs of the process are as follows,

```
javax.net.ssl.SSLSocketFactory.smali
# Android 11
vendor hash = 48:8c8rqGK0921ICcfLxviYs97wCGYu/YkQxj9e:8AOdPfLxvs97wGxj9e

# Android 11
AOSP hash = 48:8c8rqGK0921ICcfEviYs97wCGY6/YkQxj9e:8AOdPfEvs97wSxj9e
similarity = 94
```

The similarility based on the hash distance is calculated as 94, which means that the vendor class file and the AOSP class file differentiate slightly. During this study the scope of the JSSE classes being analyzed is limited to the `javax.net`, `javax.net.ssl` and `javax.crypto` sections. These classes presents the most critical components for establishing secure communication as described in section 2.3.1.

Next, in order to find the cause of the dissimilarities found in smali code a method based differential technique is used. This technique as discussed on Section 4.3.2.2 will show the added or removed methods by each vendor.

4.3.2.2 Method-based Diffing

The method-based diffing is used to examine the dissimilarities I found during the hash-based dissimilarity analysis. In Java programs, classes and methods provide

the internal structure. By focusing on method-level differences, I extract the specific modifications made to the vendor SSL/TLS stacks, providing insights into the structural changes introduced. The results of the diffing process are further analyzed to gain a comprehensive understanding of the vendor modifications and reasons behind them.

The method based differential technique will scan over each class file belonging to `javax.net`, `javax.net.ssl` and `javax.crypto` and extract the methods defined in each. The smali code indicates the method declaration by .method string. According to the code snippet in listing 4.16, the method declaration is shown as " .method public static getDefault()Ljavax/net/SocketFactory;". This gives information on the method named 'getDefault()', which is public static method which returns an object of type 'javax.net.SocketFactory'. All such defined methods which are important in secure communication [12] and also any vendor added methods are extracted in this technique.

A regex match using the pattern shown in the smali code (Listing 4.17) can be used for extracting the method signature and the method names. A query run on a vendor javax.net.DefaultSocketFactory.smali class results in the following method list.

Listing 4.17: Extraction of method definitions/names from the smali level.

```
def extract_method_names(smali_code):
  method_names = []
  pattern = r'\.method (.*?)\('

  for line in smali_code.splitlines():
      match = re.match(pattern, line)
      if match:
          method_name = match.group(1)
          method_names.append(method_name)
  return method_names

smali_file = 'javax/net/DefaultSocketFactory.smali'
smali_code = smali_file.read()

#output method list is as below
["constructor blacklist <init>", "public whitelist core-platform-api createSocket"]
```

The extraction of methods and the comparison with the baseline (AOSP) is performed as shown in the below Figure 4.7.

The diiferences between the methods that exists in the baseline AOSP class and the vendor class will present the following two families of findigs.

- Methods removed by the Android vendor

  These are the methods that only exists within the baseline AOSP build but not vendor build. These could be an indicator of features that are not deemed useful by the vendor. These removals could result in important features or security measures being removed from the TLS/SSL stack.

- Methods added by the Android vendor

  These are the methods that only exists within the vendor implementation but not within the baseline AOSP classes. These added methods could be additonal supporting/extending features for the original JSSE functions but could also result in security advancements or decrease.

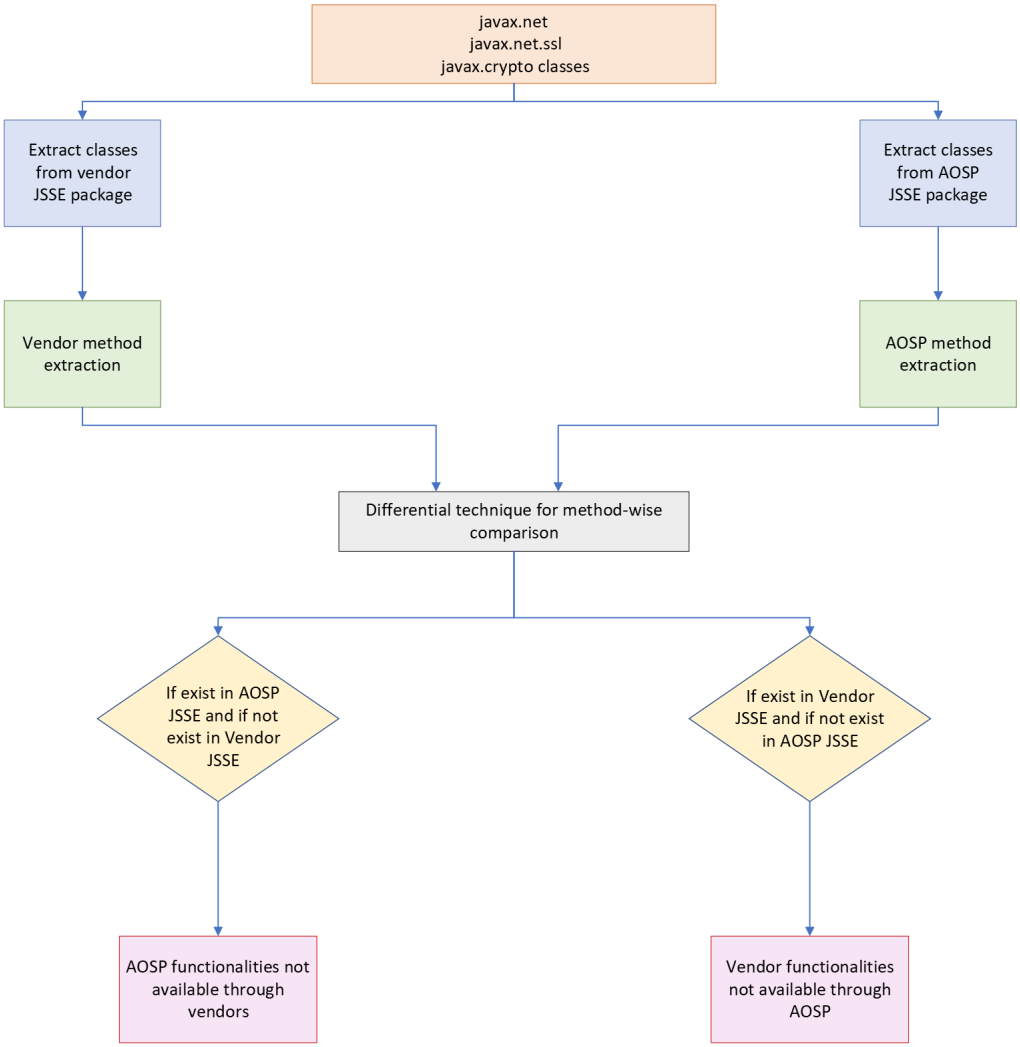


Figure 4.7: JSSE method-wise diffing technique.

4.3.3 Java Cryptographic Architecture (JCA) providers

The JCA providers are different from JSSE and JCE packages given that they are compiled into shared objects starting with C/C++ source code. These providers are stored as dynamic shared objects which means that programs can use them simultaneously and are built and stored in the Android lib or lib64 folders to be called by the programs as required.

In Android, the TLS/SSL stack is built upon the TLS protocol and cryptographic primitives provided by the cryptographic libraries. The AOSP uses BoringSSL as their default cryptographic provider but until the year 2014, OpenSSL was used as the default provider. Other open-source/proprietary cryptographic libraries can be used to replace or either increase the functionalities of the provider layer, such as LibreSSL, Libgcrypt, GnuTLS and WolfSSL.

When analyzing the vendor customizations done to shared objects, it's important to examine the source ,symbols tree of each shared object and the dependency calls between shared objects. In order to explore the vendor customizations on the TLS/SSL

Cryptographic provider	Native library shared object
OpenSSL	libssl.so libcrypto.so
BoringSSL	libssl.so libcrypto.so
LibreSSL	libssl.so libcrypto.so libtls.so
gnuPG	libgcrypt.so
GnuTLS	libmod_gnutls.so
wolfSSL	libwolfssl.so

Table 4.3: Android cryptographic providers and shared objects.

stack's native library layer, several steps of methods are used.

1. Examine the sources of the cryptographic providers.
2. Create a differentiating technique to extract the added or removed TLS/SSL protocol implementations.
3. Create a differential technique to extract the utilized cryptographic primitives and the modifications done by the vendors.

4.3.3.1 Source of Cryptographic Providers

The cryptographic provider sources can be extracted using two techniques,

- The shared object files found within the android native library layer

  The lib/ and lib64/ folders in the Android architecture includes the shared objects that is used with a particular Android OS. The names of the compiled `.so` can be used as an indicator of the cryptographic provider used. Table 4.3 shows the overview of most common cryptographic providers used.

- The `strings` command in order to identify the source code files

  According to Table 4.3 BoringSSL, OpenSSL and LibreSSL produces libssl.so and libcrypto.so as their compiled shared objects due BoringSSL and LibreSSL both being derivatives of OpenSSL. Due to this reason, a technique of differentiating among the three needs to established. The method used for this purpose within this study is examining the strings embedded within the .so files using the 'strings' command.

  - Examining the source code files

    The "strings" command's output can be filtered to extract the code paths as shown in example in listing 4.10. The following path expressions as shown in table can be used to differentiate the cryptographic provider sources.

  - Examining the versioning annotation

    As a method of validating and extending the first method of differentiating between BoringSSL and OpenSSL, the string output is filtered to extract the versioning expression as shown in the below example. For BoringSSL,

Cryptographic provider	Provider path expression
OpenSSL	external/openssl crypto/
BoringSSL	external/boringssl
LibreSSL	external/libressl

Table 4.4: Android cryptographic providers and potential source file paths.

Google does not maintain a versioning number and therefore could be used as an indicator that the shared object originates from OpenSSL.

```
1   shared_object = 'libssl.so'
2   strings_sharedObj = subprocess.check_output(['strings', file]).decode('utf-8')
3   version_info = re.findall(r'^OpenSSL.*\s\d{4}$',strings_sharedObj,flags=re.
        MULTILINE)
4
5   # output for version_info will include a version number if the origin of the
        libssl.so is OpenSSL otherwise it is BoringSSL.
6   # output expected from a OpenSSL origin libssl.so is shown.
7   ['OpenSSL 1.0.1j 15 Oct 2014']
```

Listing 4.18: Extraction of OpenSSL version using `strings` command.

4.3.3.2 Vendor Modifications on TLS/SSL Protocol Implementations

In the Android TLS/SSL stack, the protocol implementations are provided through the `libssl.so` functions. Using the nm command and applying a filter on the resulting symbol tree these functions can be extracted. Once the output of dynamic defined symbols are identied then symbol which are indicated by the symbol type 'T' can be filtered. The defined symbols with the type 'T' are the functions defined in the object file.

The function extraction is carried out on both AOSP build's `libssl.so` and vendor's `libssl.so`. BoringSSL is constantly maintained by the developers at Google and therefore the Android developers are advised to pull the latest version of BoringSSL for their products. Therefore each vendor `libssl.so` is compared with the AOSP's `libssl.so` based on the android tag embedded in the device fingerprint. This build id, points the vendor OS to the exact Android platform release.

The resulting difference could indicate the additional functions and the removed functions. Additional functionalities could indicate an added vendor function or a the latest BoringSSL function that was not part of the AOSP build. An available function within the AOSP build which doesn't include in the `libssl.so` indicates that the Android developer has removed it due to being unuseful, security concerns. Also they could have not existed (deprecated) with in BoringSSL by the time of the vendor build, therefore deprecated functions from BoringSSL should also be examined.

When considering Android over the air updates, native librabries such as BoringSSL are incorporated in to major system upates where the entire Android system image is updated. Therefore the effect of over the air updates on BoringSSL is considered negligable.

4.3.3.3 Vendor Usage of Cryptographic Primitives

The Android OS's usage of cryptographic primitives can be analyzed through mapping `libssl.so` to `libcrypto.so`. The defined functions in `libcrypto.so` is called in `libssl.so` and can be extracted using the technique visualized in Figure 4.5. For each vendor device and the related AOSP build, the mappings created are compared to find the difference between the functions being called by libssl.so in roder to implement TLS/SSL protocols.

The differences in the functions related to cryptographic primitives used to define TLS/SSL protocols, can present evidence on which functionalities developers use within their TLS/SSL stack and the functionalities the developers remove during their modifications.

4.3.4 Java Cryptographic Extension (JCE) packages

The current Android implementation includes two main JCE providers, Conscrypt and OkHttp. Conscrypt was built and currently maintained by Google while OkHttp is an externally sourced HTTP/HTTPS client sourced by Square. In order to study the Android supply chain effect on Conscrypt a similar method as the JSSE packages is used. The baseline AOSP build is compared with the vendor's Conscrypt adaptation in order to characterize additional functionalities and removed functionalities within the vendor implementation. As the Conscrypt module is built upon the JCA providers and connects the providers to the JSSE, the prior step of supply chain effect on JCA providers can be complemented through analyzing the effect of developer modifications on the bridge between JCA and JCE. After the extraction of the modifications done by vendors on Conscrypt classes, the official Conscrypt release by Google will be used to study the final impact of these findings.

Conscrypt's job within the Android TLS/SSL stack is to act as a wrapper between JSSE and JCA, therefore by practice the function names defined in Conscrypt is similar to the ones found within OpenSSL. If any functionality revocation or additions that was found during the

The externally-sourced JCE components OkHttp and BouncyCastle require a higher-level study on the vendor specific variations before examining the modificationn to the intended functionality. Therefore in this study, a breif study on the OkHttp and BouncyCastle libraries is conducted in order to gather information on the existence of these libraries in the wild.

Chapter 5

Results and Discussion

The empirical results obtained as a result of applying my research methodology are organized in dedicated sections that reflect the customizations done to each layer of the TLS/SSL stack with regards to the baseline, and how these changes affect the functionalities and security guarantees intended by the AOSP source code. Specifically: Section 5.1 analyzes vendor modifications on the JSSE layer, separating the analysis into added functionalities and removed ones. Section 5.2 presents the analysis of JCA providers. Lastly, Section 5.3 analyzes interface between JSSE and the JCA providers, including the JCE packages.

5.1 JSSE Customizations

After decompiling the Java Secure Socket Extension (JSSE) packages as described in Section 4.2, the extracted smali representations of the packages are analyzed using the methods described in Section 4.3.2. I use a fuzzy hashing technique, and the method-wise differentiating analysis to study the effect of the Android supply chain on the JSSE implementation.

Throughout this Chapter, I rely on Ssdeep to compute the dissimilarities of vendor JSEE, JCA and JCE implementations with regards to their corresponding baseline. Ssdeep [59] is a fuzzy hashing tool which uses the CTPH technique to calculate two hash values for each smali class. The first hash value is generated using the vendor's JSSE smali class and the second one using the smali class of its corresponding AOSP baseline version. The hashes obtained are then compared to determine the similarity between the two. Their similarity score ranges from 100 (complete similarity) to 0 (total lack of similarity). The similarility score highlights the smali classes which are similar to each other, therefore in order to obtain an overall view on the modifications done by each vendor, we compute the average **dissimilarity score**. The dissimilarity score is calculated between the JSEE packages and their corresponding baseline, for all the devices for a given vendor and Android version; i.e., the average of dissimilarities for `javax.net` and `javax.crypto` classes of each vendor and version.

Figure 5.1 shows the results of the average dissimilarity for each vendor.

While a small cluster of vendors such as Zebra, Oneplus and Blackshark keep the original AOSP source code for their own Android distributions, 67% of the vendors

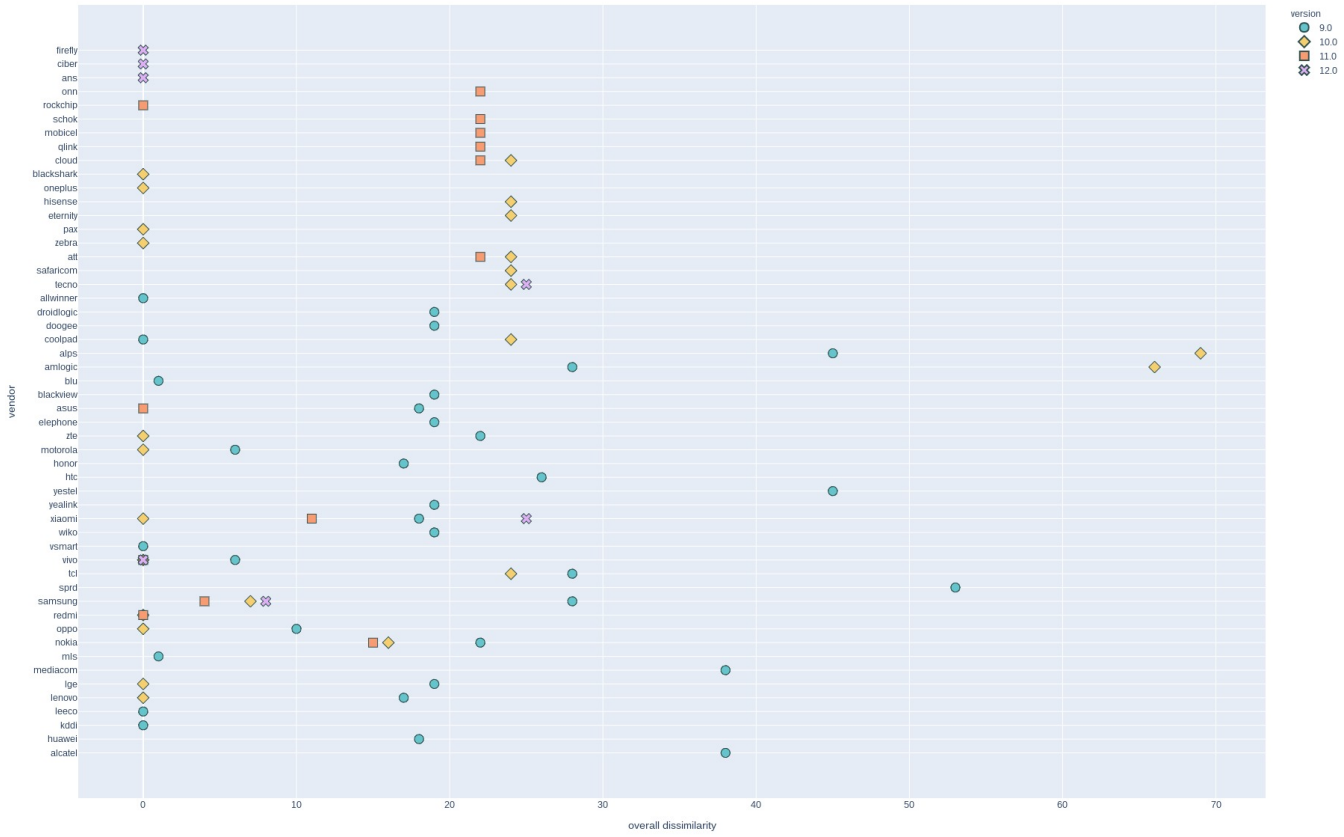


Figure 5.1: Android JSSE package dissimilarities between each vendor device and the baseline AOSP build.

and versions in my dataset diverge from their AOSP counterpart. Examining Figure 5.1, it is noticeable that some vendors present similar patterns in their changes and customizations. For example, all Nokia, ZTE, Qlink, Cloud, Mobicel, Schok, Onn and AT&T devices that runs on Android 10 have the same dissimilarity score. The observed alikeness can be further analyzed by examining the dissimilarities of each JSSE class of the vendor. For the Android 10 devices with the same average dissimilarity, the class-wise dissimilarity of the `javax.net.ssl` sub package can be visualized in Figure 5.2.

This could suggest the presence of a common node in the Android supply chain behind these vendors or common development approaches. In fact, publicly available information on the web suggests that some of the identified similarities arise from existing partnerships and device factories building devices for other brands. This commonality is studied further in Section 6.1, yet we show two examples below:

- **Nokia and ZTE.** These two brands have joined forces with the OSSii (Operations Support Systems interoperability initiative). This supports easier interoperability between operating support systems and reduces the overall cost of updates and time-to-market. This common stakeholder could potentially be the reason for the JSSE API similarities detected through the analysis.
- **ZTE,Qlink and AT&T.** ZTE has entered into partnerships with both mobile net-

work operators in order to provide additional functionalities to their end users. The network stack commonalities seen during the analysis could be due these potential partnerships.

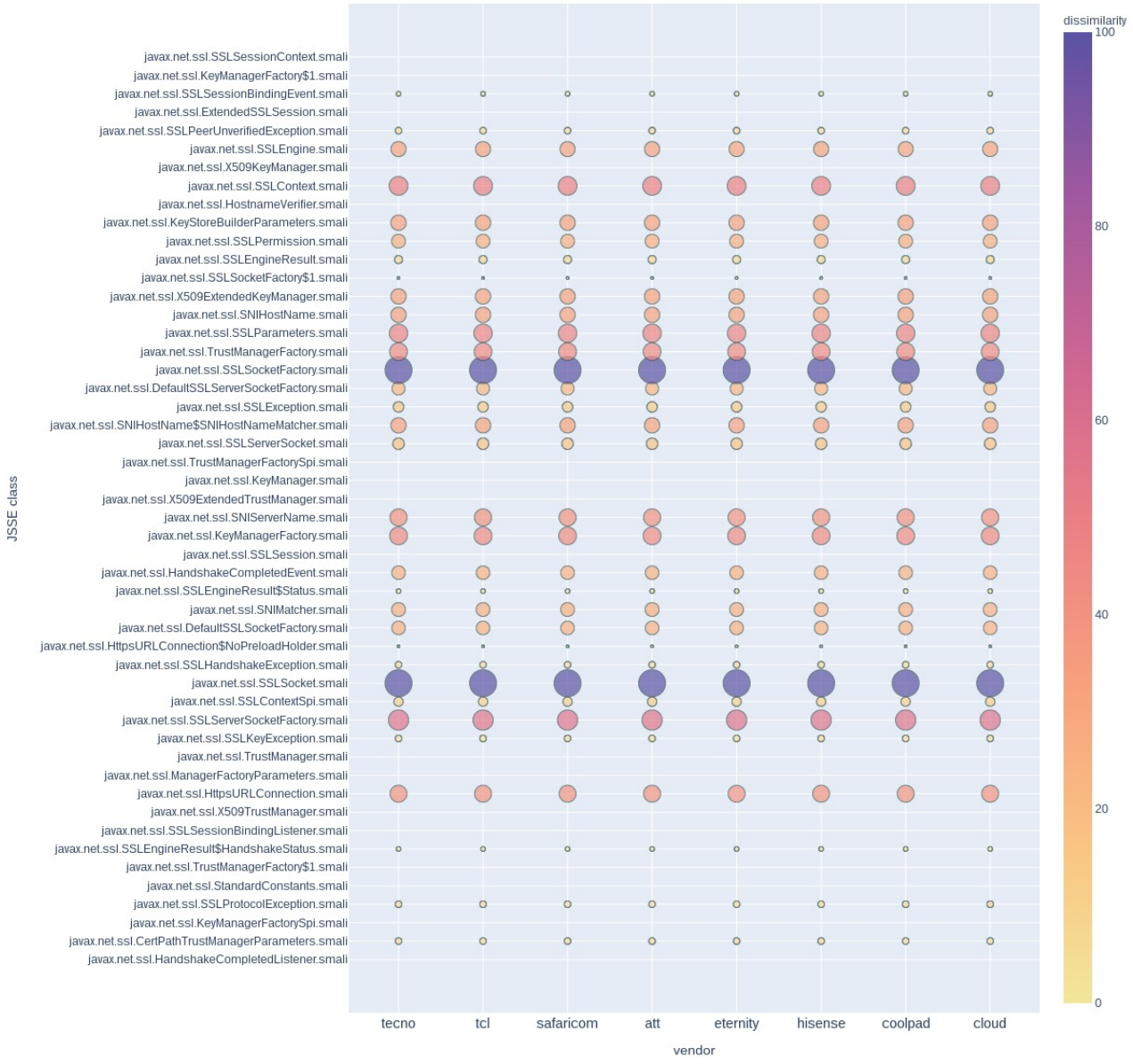


Figure 5.2: Vendor JSSE classes with akin dissimilarities compared to the baseline JSSE classes.

Table 5.1 shows the vendors with the highest dissimilarity scores. The highest dissimilarities are found in Alps and Amlogic (chipset manufacturer) devices: 69% and 66%, respectively.

Android vendor	Android version	overall dissimilarity
Alps	10.0	69
Amlogic	10.0	66
Sprd	9.0	53
Alps	9.0	45
Yestel	9.0	45
Alcatel	9.0	38
Mediacom	9.0	38
Amlogic	9.0	28
TC;	9.0	28
samsung	9.0	28

Table 5.1: JSSE package dissimilarities across vendors and versions for the vendors displaying most deviations from the baseline AOSP.

In order to examine these deviations, we analyze the structural difference of these classes across vendors and versions. This is done using a method-level differential technique of the JSSE class code with regards to the AOSP baseline. This approach allows identifying additional functionalities (i.e., new methods) and removed ones (i.e., removed methods) across devices.

5.1.1 Removed JSSE functionality

Figure 4.7 shows the pipeline that I designed to identify JSSE methods removed by the vendors. The total number of methods removed from the vendor JSSE packages are shown in Figure 5.3. The removed methods (in y-axis) indicates the number of unique methods removed/missing from the inspected JSSE classes. Vendor devices (in x-axis) are categorized into Android versions using different symbols and colors; blue, red , green and purple respresenting version 9, 10, 11 and 12 respectively. Alps, Sprd and Amlogic vendors display a significant deviations from the AOSP baseline JSSE classes. In Alps devices running Android version 10, I identify up to 40 methods in total been removed. This accounts for 7% of the intended AOSP JSSE functionalities. In the case of Sprd devices running on Android 9, 5% of the JSSE functionalities are removed. It is significant to note that while the mentioned vendors are not among Android certified vendors, Samsung devices —which is an Android ceritied vendor—also have method removals, though minimal.

For reference, note that the JSSE AOSP code for Android versions 9, 10, 11, 12 includes 636 different methods in total when considering all the JSSE sub-classes belonging to `javax.net` and `javax.crypto`.

Figure 5.4 categorizes the removed methods discovered in the previous step into the JSSE classes they belong to. This provides insight into the most affected JSSE classes from vendor modifications.

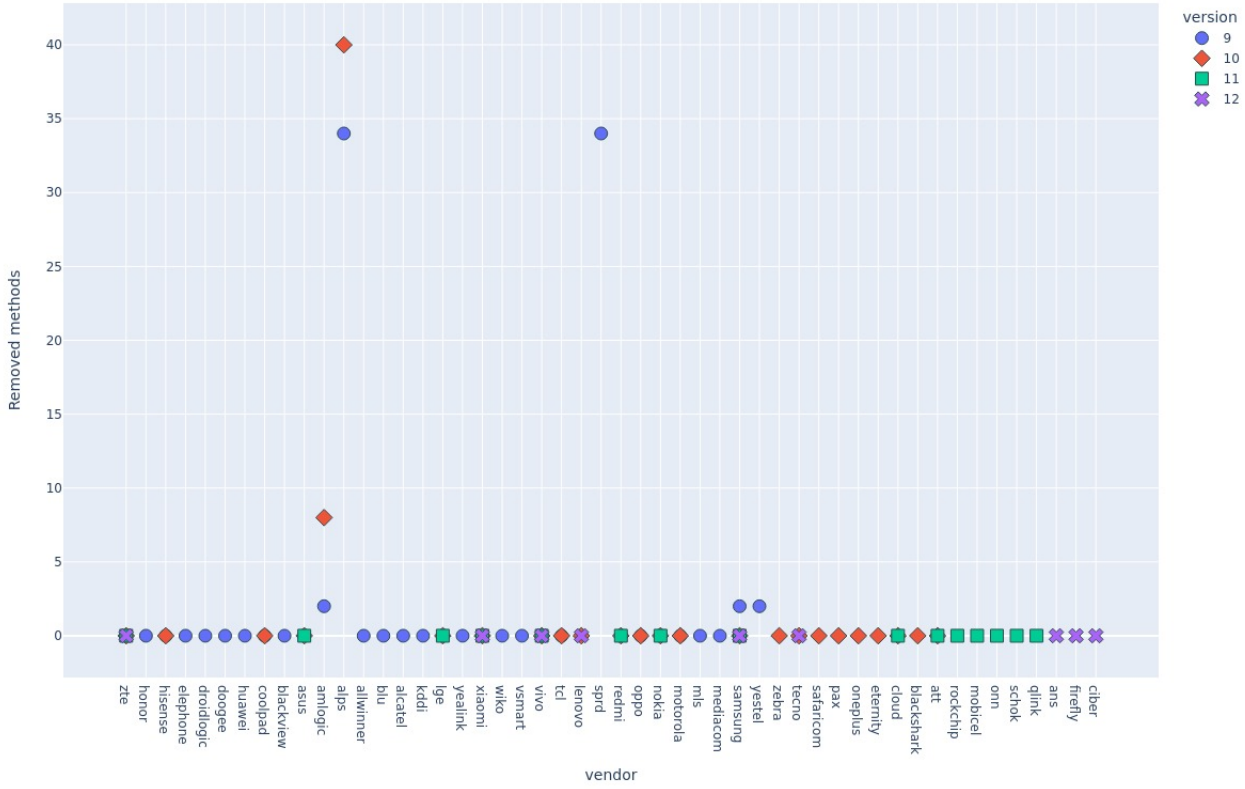


Figure 5.3: Android JSSE methods removed from version 9,10,11,12 vendor devices, using method-wise diffing.

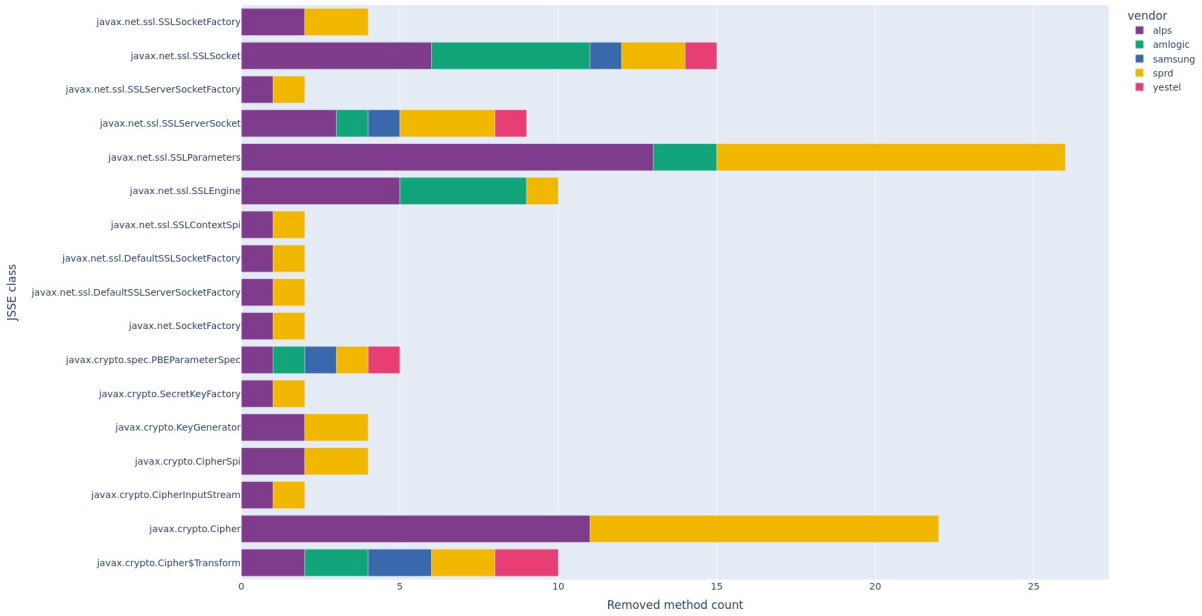


Figure 5.4: Android JSSE removed methods characterized into their JSSE classes.

Figure 5.4 shows some of the detected JSSE classes with removed methods in Alps, Sprd, Samsung and Amlogic devices. I discuss a few cases of interest next, using the information provided in the oracle OpenJDK documentation along with the Android documentation:

- The `javax.net.ssl.SSLParameters` is heavily altered by these vendors. This class holds a key role in TLS security as it allows configurations for secure communication such as defining the accepted cipher suites and also endpoint identification algorithms for the SSL/TLS handshake, server name indications, protocol list that is allowed in SSL/TLS communication etc.
- Up to 11 public methods are removed from the class `javax.crypto.Cipher` class. This class is fundamental for providing functionalities for cryptographic ciphers, specifically encrypting and decrypting operations. In fact, this class sets the foundation for the JCE framework.

In order to trace method removal, Figure 5.5 shows the removed methods in the final leaf node, while the parent nodes depict the `javax.net` or `javax.crypto` class paths leading to the discarded functionalities.

The color of the leaf node interpret the number of unique vendor and versions with the removed functionality. Their corresponding usage or purpose are described in Table 5.2. Some of the functionalities removed are critical for secure communication establishment through TLS/SSL. Among others, these methods allow developers to set prioritized algorithms and providers, while trusted server names are meant to provide app developers with mechanisms to strengthen the security of network communication.

Javax class path	Removed method	Removed method functionality	Vendor:Version
	clone	create copies of the sslparameter object which can be modified without altering the original sslparameter object.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getAlgorithmConstraints	returns the cryptographic algorithm constraints that are set by the developer, this includes ciphersuites, signature algorithms, key sizes.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getApplicationProtocols	returns an array of prioritized application layer protocols that can be used to negotiate during SSL/TLS protocols.	alps:10.0 amlogic:10.0
	getEndpointIdentificationAlgorithm	returns the endpoint identification algorithm, which specifies how the client device (android device) should verify the server hostname.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getSNIMatchers	returns the list of rules for server name indication matching that occurs during the SSL/TLS handshakes.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getServerNames	returns the servers names (server name indications) set for SSL/TLS connections. These are the host names desired by the clients for SSL/TLS connection.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getUseCipherSuitesOrder	returns a boolean value indicating if the enabled cipher suites order is followed SSL/TLS handshake.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	setAlgorithmConstraints	set the algorithm constraints for the SSL/TLS handshake, these could be additional to the constraints set during runtime. These constraints include cryptographic algorithms, key , key size and if not set then no validation will be carried out during the handshake.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	setApplicationProtocols	sets the list of prioritized application layer protocols that can be used to negotiate during SSL/TLS protocols. The protocols configured using this method are compared to what the target device sends. If the protocols are not matched with any of the set ones, then alternative actions are taken or the connection is terminated.	alps:10.0 amlogic:10.0
	setEndpointIdentificationAlgorithm	sets the endpoint identification algorithms for the SSL/TLS handshakes which is by default https. This functionality is critical for preventing man-in-the-middle attacks. If not set then the host name verifier will use the predetermined algorithms provided by the SSL/TLS implementation.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0

Results and Discussion

Javax class path	Removed method	Removed method functionality	Vendor:Version
javax.net.ssl.SSLParameters	setSNIMatchers	sets the server name indication matchers (patterns) used for establishing the SSL/TLS connection. This allows the client to provide the hostname rules when it's trying to connect to during the handshake, therefore allowing the server to present the digital certificate for the matching hostname.This function is important when a server hosts multiple services (for example websites) with different domain names but through the same IP address. If the SNI matchers are not presented then the server wouldn't recognize which digital certificate to present. This could lead to potential certificate mismatches.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	setServerNames	sets the list if host names (server names) that the client Android device is expecting to match during the SSL/TLS handshake. Although similar to setting SNI matchers this allows the explicit specification of server names.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	setUseCipherSuitesOrder	sets whether the list of prioritized ciphersuites order should be followed, the list of ciphersuites are set using the setCipherSuites() function.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.net.ssl.SSLSocket	toString	Converts the SSLSocket object to a string, this is mostly used for logging and debugging purposes.	alps:10.0 alps:9.0 alps:9.1 amlogic:10.0 amlogic:9.0 samsung:9.0 sprd:9.0 yestel:9.0
	getHandshakeSession	returns the SSL/TLS handshake currently being established. The results include the negotiated cipher suites and protocol versions and can be retrieved before the handshake is fully completed.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getApplicationProtocol	returns the application protocol value negotiated for the current connection, if there are any protocol-specific logic to be implemented then this string output is valuable to the developer,these also include the protocol prioritization list and how it is processed.	alps:10.0 amlogic:10.0
	getHandshakeApplicationProtocol	returns the application protocol which is negotiated during the SSL/TLS handshake before it is completed, the outputs can be run against the prioritized application protocol list.	alps:10.0 amlogic:10.0
	getHandshakeApplicationProtocolSelector	returns the application protocol selector which is responsible for selecting the application protocol that is to be used during the handshake to be established.	alps:10.0 amlogic:10.0
	setHandshakeApplicationProtocolSelector	allows to customize the protocol selection logic of the SSL/TLS handshake. This can be used to override the SSLParameter.setApplicationProtocols. The resulting string is either the application protocol name or an empty string or a null. The list parameter of this method allows the configuration of the application protocol names.	alps:10.0 amlogic:10.0
javax.net.ssl.SSLServerSocket	toString	converts the SSLServerSocketobject to a string; this inclides local address and the port number of the socket.	alps:10.0 alps:9.0 alps:9.1 amlogic:10.0 amlogic:9.0 samsung:9.0 sprd:9.0 yestel:9.0
	setSSLParameters	sets the SSL/TLS parmeters for the new connection, the SSL parameters include; getCipherSuites(), getProtocols(), getNeedClientAuth(), getServerNames() and getSNIMatchers().	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getSSLParameters	returns the SSL/TLS parameters for the new connection.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.net.ssl.SSLEngine	getHandshakeSession	returns the information on the ongoing SSL session. This is useful when the protocols being negotiated during the initialization needs to be accessed.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getApplicationProtocol	returns the application protocol value negotiated for the current connection, if there are any protocol-specific logic to be implemented then this string output is valuable to the developer,these also include the protocol prioritization list and how it is processed.	alps:10.0 amlogic:10.0
	getHandshakeApplicationProtocol	returns the application protocol which is negotiated during the SSL/TLS handshake before it is completed, the outputs can be run against the prioritized application protocol list.	alps:10.0 amlogic:10.0
	getHandshakeApplicationProtocolSelector	returns the application protocol selector which is responsible for selecting the application protocol that is to be used during the handshake to be established.	alps:10.0 amlogic:10.0
	setHandshakeApplicationProtocolSelector	allows to customize the protocol selection logic of the SSL/TLS handshake. This can be used to override the SSLParameter.setApplicationProtocols. The resulting string is either the application protocol name or an empty string or a null. The list parameter of this method allows the configuration of the application protocol names.	alps:10.0 amlogic:10.0
	getSecurityProperty	not a defined method by Java SE. Google's implementation of SSL Socket Factory to extract the security property of ssl.SocketFactory.provider which includes the set SSLSocketFactory to be used.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0

Javax class path	Removed method	Removed method functionality	Vendor:Version
javax.net.ssl.SSLSocketFactory	log	used for debugging the SSL socket connection being established.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.net.ssl.SSLContextSpi	getDefaultSocketsetDefault	added by Google for retrieving the default socket in the source code, this is a private method used for obtaining the parameters for the SSL socket created.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.net.ssl.SSLServerSocketFactory	log	marked as 'android-added' in the Google AOSP code, for debugging purposes.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.net.SocketFactory	setDefault	marked as 'android-added' in the Google AOSP code, for testing purposes. Requires developers to remove it during the final build.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.crypto.Cipher	checkCipherState	validates the state of the cipher in order to successfully carry out update or final encryption or decryption operations. The cipher should be successfully initialized and in either ENCRYPT_MODE or DECRYPT_MODE.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	checkOpmode	checks the operation mode of the cipher, the op mode could be either ENCRYPT_MODE, DECRYPT_MODE, WRAP_MODE or UNWARP_MODE. Used for initializing the cipher object with logical conditions based on the op mode.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	chooseProvider	chooses the provider that is able to support the ongoing cryptographic operation based on the op mode, key, and algorithm parameters. This is significantly important given that Java applications can iterate over the available providers to find the best fit for their requirements.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	createCipher	returns the cipher that fulfills the required transformation.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getAlgorithmParameterSpec	returns an AlgorithmParameterSpec object containing the maximum cipher parameter value according to the jurisdiction policy file, which is used to verify the algorithm parameters associated with a cryptographic object.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getOpmodeString	defined in the AOSP source code but it is a private method that is not called inside the class. Potentially added for testing purposes.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	tokenizeTransformation	performs the cryptographic transformation of the string using the algorithm name, feedback technique, and padding.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	updateProviderIfNeeded	if the provider selection using the first provider is not upto date and can't perform the required operations then the registered provider is updated or a compatible alternative is set.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.crypto.CipherSpi	getTempArraySize	used by the bufferCrypt functionality to determine the array size to hold the temporary data during the data transformation.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.crypto.KeyGenerator	disableFailover	disables any failover mechanisms or re-initialization processes, used when the provider is set successfully.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	nextSpi	updates the current active cryptographic service provider interface and return the next available spi incase of failover extending the usability of other Spi providers.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.crypto.spec.PBEParameterSpec	getParameterSpec	returns the password-based encryption (PBE) operation's cipher algorithm parameter specifications such as the salt value and iteration count. The results can be used to configure the PBE algorithms to perform additional security measures.	alps:10.0 alps:9.0 alps:9.1 amlogic:10.0 amlogic:9.0 samsung:9.0 sprd:9.0 yestel:9.0
javax.crypto.CipherInputStream	getMoreData	used to read the data from the input stream and process it using the Cipher object set and then store the transformed data in an output buffer.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.crypto.SecretKeyFactory	nextSpi	updates the current active cryptographic service provider interface and return the next available spi in case of failover extending the usability of other Spi providers.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0

Table 5.2: The JSSE functionalities removed by different vendors based on method names.

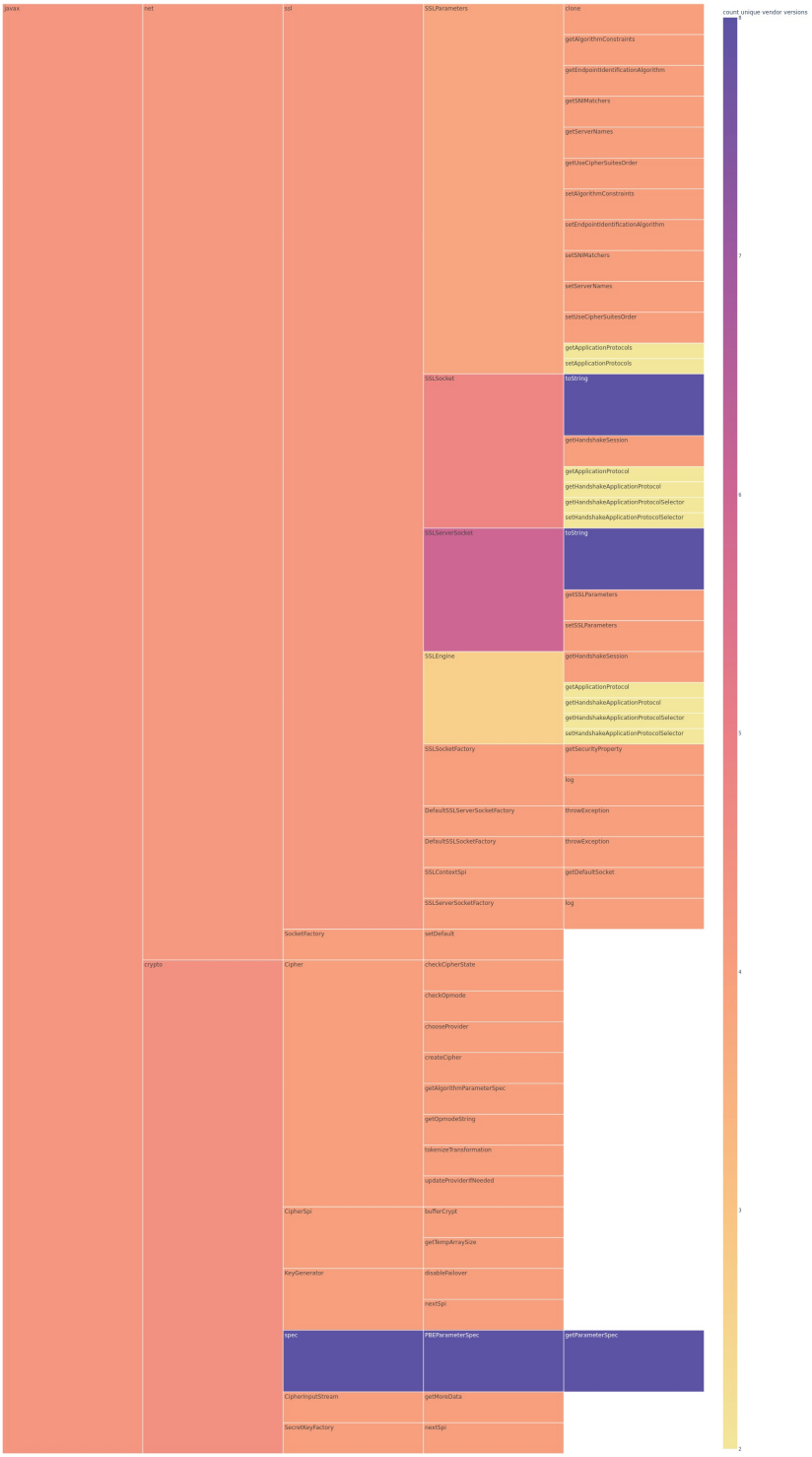


Figure 5.5: JSSE functionalities removed by vendors classified into each class. The colorscale depicts the number of unique vendor:versions these removals are detected in.

This analysis also reported methods not present on AOSP, so presumably added by the vendors. Section 5.1.2 discusses the additional functionalities detected through

the method-wise diffing technique. Examining the added functionalities brings an opportunity to detect if removed functionalities are caused by typos or renames. My analysis shows that 6 methods that are flagged as removals through my pipeline are actually replaced by alternative methods in Alps and Sprd devices, these findings are futher discussed in subsection 5.1.2.

5.1.2 Added JSSE functionality

Figure 5.6 depicts the overall number of methods added by the vendors compared to their corresponding baseline AOSP build. Similar to the results shown in the removed methods analysis (Section 5.1.1) Alps and Sprd vendors both show deviations from the rest of the Android vendors by incoporating 12 additional methods to their JSSE classes. In terms of the original JSSE classes they belong to, the altered class distribution (Figure 5.7) in terms of additional functionality shows that javax,crypto.Cipher is mostly altered in Alps version 10 and Sprd version 9 devices.

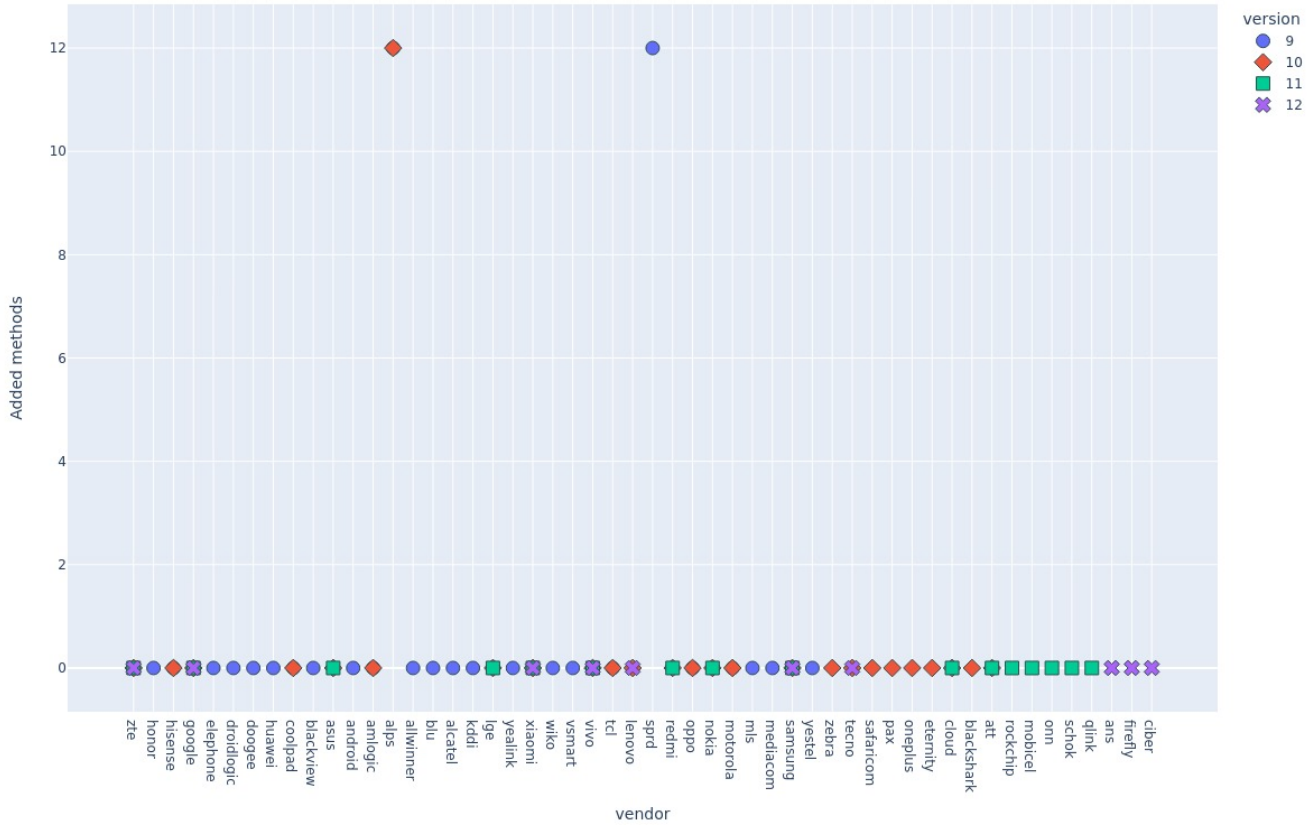


Figure 5.6: Android JSSE methods added for version 9,10,11,12 vendor devices, using method-wise diffing.

In order to further analyze this behavior of vendors and to verify whether the added methods compensate for the removed functionalities, I use the method-wise logic comparison approach. Figure 5.8 shows the methods and the JSSE path which are added by the vendors. The color of the leaf node interpret the number of unique

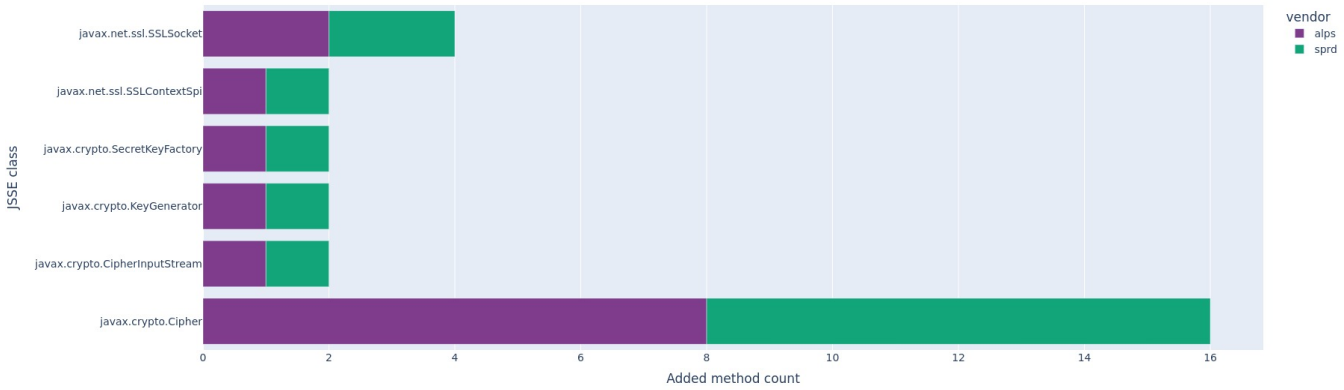


Figure 5.7: Android JSSE added methods characterized into their JSSE classes.

vendor and versions with added functions. Their corresponding usage or purpose are described in Table 5.3.

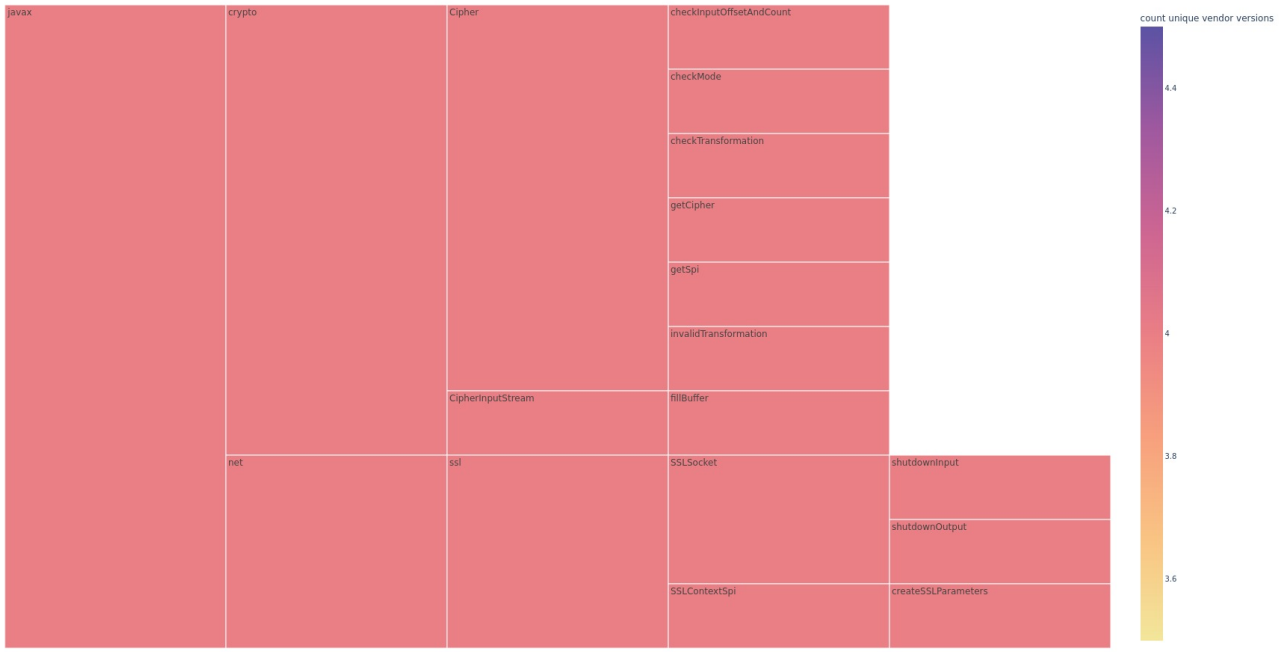


Figure 5.8: JSSE functionalities added by vendors classified into each class. The colorscale depicts the number of unique vendor:versions the additions are detected in.

Unfortunately, since none of the added methods is documented, the usage analysis forced me to manually reverse the vendor-added functions to identify their purpose. The result of the manual analysis (Table 5.3) suggests that, while some of the lost functionalities due to method removals are compensated with the addition of alternative methods, some critical features such as endpoint verification, hostname veri-

fications are missing from vendor TLS/SSL stacks entirely.

Javax class path	Added method	Added method functionality	Vendor:Version
javax.crypto.Cipher	checkInputOffsetAndCount	This method is taken from 'libcore/luni' which includes Android's implementation and is incorporated into the OpenJDK's class file. This is used to check for the input stream length and validates the offset value.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	checkMode	Similar to the above function, this is adapted from the Android's implementation of secure communication. Serves the same purpose as checkOpmode.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	checkTransformation	checks the transformation string for algorithm, mode and padding similar to tokenizeTransformation, again adapted from Android secure communication implementation	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getCipher	returns the instance of javax,crypto.Cipher using the trasnformation string and provider. This is a similar to getCipher in AOSP javax.crypto.Cipher class.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	getSpi	performs actions similar to the AOSP method updateAndGetSpiAndProvider.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	invalidTransformation	provides an exception handling for the checkTransformation. Completes the functionality provided by AOSP tokenizeTransformation	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.crypto.CipherInputStream	fillBuffer	stores the transformed data from the cipher operation in the output buffer similar to the getMoreData() in AOSP	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.net.ssl.SSLSocket	shutdownInput	used to indicate that the input stream should be be shut down due to unsupported it being an operation, potentially implemented to avoid compatibility issues	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
	shutdownOutput	used to indicate that the output stream should be be shut down due to unsupported it being an operation, potentially implemented to avoid compatibility issues	alps:10.0 alps:9.0 alps:9.1 sprd:9.0
javax.net.ssl.SSLContextSpi	createSSLParameters	uses the as-is ciphersuites and protocols from the SSLSocket and sets them as the SSLParameters.	alps:10.0 alps:9.0 alps:9.1 sprd:9.0

Table 5.3: The JSSE functionalities added by vendors based on method names and their functionalities.

The methods added into the `javax.crypto.Cipher` and `CipherInputStream` classes in Alps and Sprd devices are carried over from the `libcore/luni` folder. These were included in the Google implementation of the javax classes, but these classes have been replaced by the OpenJDK sourced javax classes since the release Android 7.0 releases [2, 3]. This could be an indication of these vendors using old JSSE class implementations that they used for their older Android releases (Android version 7 and earlier). From a developer point of view, these additonal methods are not part of the official API documentations provided by Oracle, OpenJDK or Google. Therefore application developers might not be aware of their existence, which could cause program malfunctioning and degraded security.

5.1.3 Functionality and Security Loss Overview

This subsection puts in context both method removal and additions. The diffing results on the JSSE API level (Tables 5.2 and 5.3) show that the methods used to validating and identifying the endpoints such as setAlgorithmConstraints, setEndpointIdentificationAlgorithm, setSNIMatchers, setServerNames, and setUseCipherSuitesOrder are not found in Alps and Sprd devices. I will discuss the importance of these missing methods and the impact of removing them below.

- **Algorithm constrainst and configurations:** Method `setAlgorithmConstraints` is intended to set algorithm constraints for certificate validations, removal of this

functionality limits the ability to improve the security of the SSL/TLS communications by enforcing specific cryptographic algorithms or key sizes for certificates.
End point algorithms defined through the `setEndpointIdentificationAlgorithm` method are used to determine how the hostname presented in the digital certificate is verified against the actual hostname of the server. Disabling this ability weakens end-point security by preventing it from performing hostname verification. Consequently, it increases the risk of successful man-in-the-middle (MITM) attacks. When this functionality is not used as intended by the Operating System, the default endpoint identification algorithm provided by the SSL/TLS protocol implementation is used (implemented by the JCA providers). This default algorithms might not provide the acceptable level of security as a customized, more meticulous algorithm. This weakened point within the TLS/SSL handshake process could provide an external entity (observer) the opportunity to intercept the communication between the client and the servers and then present different certificates that are not valid for the actual hostname. In other words, it allows performing man-in-the-middle (MITM) attacks by, for example, state surveillance agents. This could lead the Android client relying on the default stack to open a communication channel with the attacker's man-in-the-middle point or a malicious target, which compromises the confidentially of the client communication.

- **Hostname verifications:** The absence of the `setServerNames` and the `setSNIMatchers` prevents the client from performing additional verifications during the TLS handshake. Setting server name indications (SNIs) gives client the control over hostname verification based on the server SNI extension and setEndpointIdentificationAlgorithm server names provide the security mechanisms to specify the expected hostnames explicitly. If the target host is part of a virtual hosting system where multiple hostnames are hosted on the same IP address, the identification of the exact target will rely on the SNI extension. If SNI patterns are not set then server might not be able to properly select the correct certificate to present therefore resulting in certificate mismatches but also give the ability to present fraudulent certificates leading to unauthorized access.

- **Ciphersuites prioritizations:** The order of the client preferred ciphersuites is key to enhance the security of TLS communications.[77] In the JSSE API, this is invoked through the `setUseCipherSuitesOrder` method. Clients can decide the order and remove ciphersuites that can be potentially vulnerable [77], to enhance the security of their programs by invoking the setUseCipherSuitesOrder functionality. Otherwise, the default ciphersuites (and order) will be used. Removing this functionality prevents developers from selecting their preferred ciphersuites for enhancing the security of their communications with the cloud.

These removals can have a significant impact on popular TLS security mechanisms such as certificate pinning. This is an additional security mechanism taken by app developers to enhance the security of their encrypted communications on the endpoint. Pinning enables developers to validate server certificates during the client-server communication establishment [88]. Traditionally, the Android client will rely on the chain of trust through the CAs to perform the certificate validation, many of which can be potentially malicious or forged. Yet, certificate pinning gives the application an additional opportunity to specify a set of trusted certificates or public key

hashes. Removing methods such as setEndpointIdentificationAlgorithm, setServerNames and setSNIMatchers prevents Android developers from performing this action using default Android APIs. This platform fragmentation and the lack of trust on Android vendors is one of the driving forces pushing mobile app developers to integrate third-party TLS libraries on their code.

5.2 JCA Providers Customizations

Android manufacturers and vendors also have the freedom to chose their preferred cryptographic provider like OpenSSL or BoringSSL. These providers offer low-level protocol implementations and cryptographic primitives that are later used by protocol implementations. The customizations done on this layer begins with the addition of the native source code into the appropriate folder within the source code. Usually this location is either the `external/` folder or the `vendor/` folder, and the C/C++ source should be accompnied by the Makefile or a CMake file. The vendor is free to incorporate any changes to the existing code by AOSP or add an alternative provider in place of BoringSSL. The build system (Section 2.4) then compiles the source code, which converts the native code into objects files which are then linked with the dependency object files to create shared objects. After a successful compilation, the vendors can use the customized provider or the alternate provider shared objects (in `system/lib` or `vendor/lib`) exist as the source of cryptographic primitives and protocols.

As mentioned in Section 4.3.3, the source of each shared object (i.e., `libssl.so` and `libcrypto.so`) can be used to infer and study which open-source implementation or derivative of OpenSSL is used as the cryptographic provider in a given device. It is important to note that some devices may include more than one cryptographic provider on their firmware, either for extending their functionality or for application compatibility reasons. For example, due to the minimal approach followed by BoringSSL, vendors might gravitate towards using OpenSSL as a secondary provider [40].

5.2.1 Provider Choice

The choice of cryptographic provider of a vendor can depend on their required use cases. In order to detect the usage of different cryptographic providers in the Android ecosystem, I conduct the source analysis on the provider shared objects `libssl.so` and `libcrypto.so` (Section 4.3.3.1). The source analysis of the shared objects shows that, while the majority of `libssl.so` and `libcrypto.so` shared objects within the collected dataset are compiled using BoringSSL, some devices still rely on OpenSSL as their default cryptographic providers as shown in Figure 5.9. The default OpenSSL implementation has had its share of severe vulnerabilities over the years such as Heartbleed [47] and DROWN [84]. Heartbleed was a critical vulnerability discovered in OpenSSL during 2014. This weakness allowed attackers to exploit a flaw in the heartbeat extension in the SSL/TLS protocol. By sending a malicious heartbeat request, the attacker got exposed to the sensitive information such as private keys, username indicators and passwords. This caused a major switch in the Android secure communication implementation, given that AOSP started the replacement process of OpenSSL with BoringSSL after the disclosure of Heartbleed. Since 2014, OpenSSL is still the subject of many vulnerabilities, with 13 vulnerabilities being dis-

losed in the year 2023 alone. Therefore if OpenSSL is used in place of BoringSSL, it pivotal that vendors follow proper maintainance on their OpenSSL repository.

Figure 5.9 provides the overview of the results from the provider source analysis. Each vendor and their versions are mapped to the cryptographic providers detected in their SSL/TLS stacks. Within the firmware images included in the dataset, I found three cryptographic providers; BoringSSL (indicated in blue), OpenSSL (indicated in green) and Libgcrypt (indicated in pink). Figure 5.9 covers Android versions spanning from version 7 to the latest android version. The OpenSSL to BoringSSL switch occurred during the release of Android 7, therefore this provides with an opportunity to navigate the success of the Google's initiative to switch their own derivative of OpenSSL. Next, I discuss in depth vendors' cryptographic provider choices for Libgcrypt and OpenSSL.

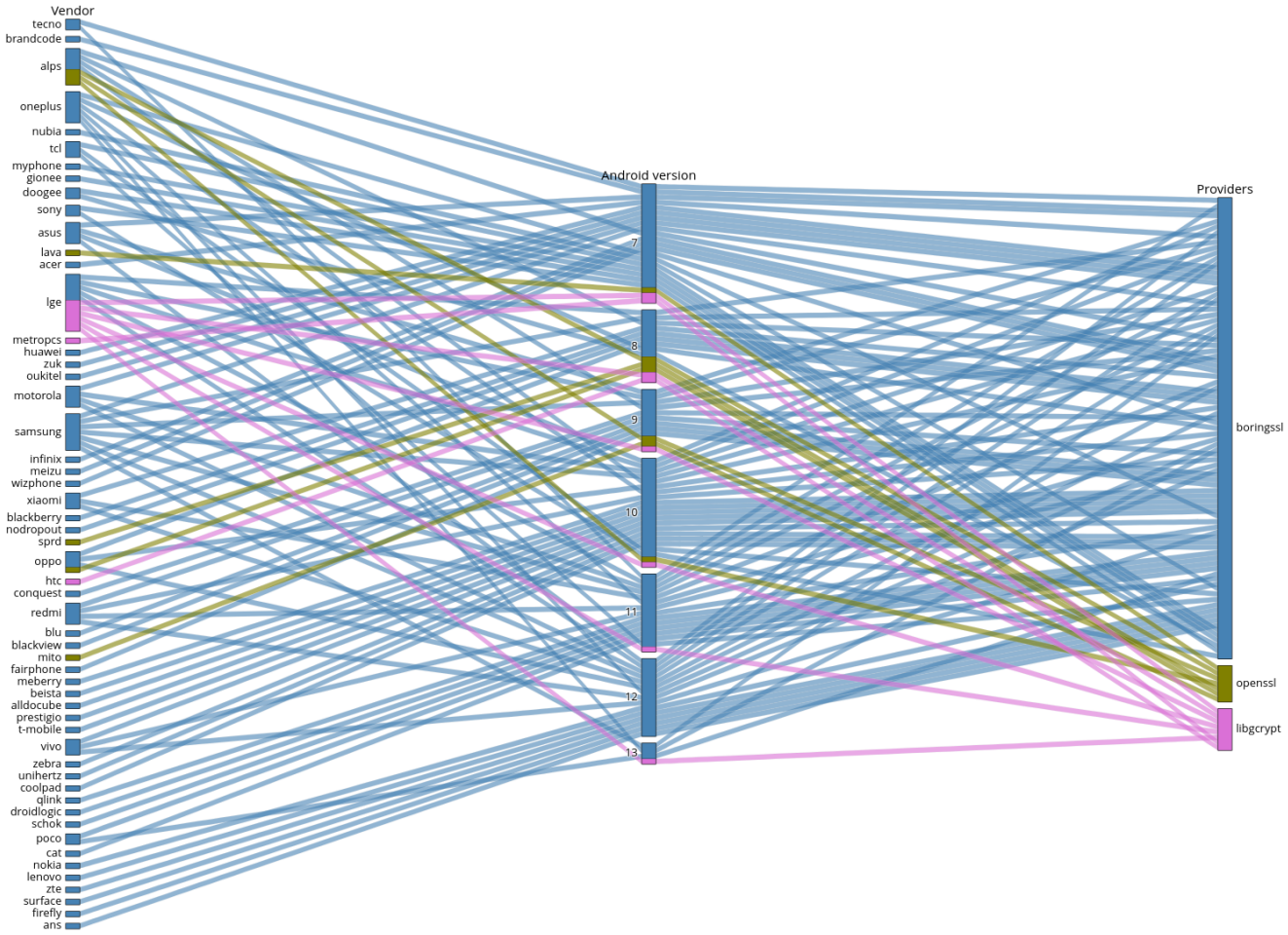


Figure 5.9: Android TLS/SSL provider distribution by vendor and version.

Libgcrypt

Figure 5.9 shows that LG electronics (South korean Android device manufacturer), Metro (a North American virtual MNO owned by T-mobile), HTC (chinese manufac-

turer which provides ODM services to Google) incorporate `libgcrypt.so` into their TLS/SSL stack, which is another OpenSSL alternative that hasn't removed as much functionalities from the original OpenSSL compared to BoringSSL.

According to the analysis results, Libgcrypt and BoringSSL are both used in LGE from versions 7 to 13. This combination could be attributed to Libgcrypt offering a specific feature that is not present in BoringSSL but is required by the vendor. Metropcs and HTC devices running on Android 7 and 8, solely rely on Libgcrypt as their cryptographic provider. Libgcrypt has demonstrated a higher level of security compared to alternatives such as OpenSSL. Notably, the last publicly disclosed vulnerabilities in Libgcrypt date back to 2021, with only three identified vulnerabilities, none of which were classified as severe [25].

OpenSSL

Five of the 55 Android vendors considered in our analysis include OpenSSL as their cryptographic provider. Lava (Android ceritified vendor, manufactured in India), Sprd, and Mito (manufactured in Indonesia) vendors include OpenSSL as their default cryptographic provider while Alps and Oppo use OpenSSL along with BoringSSL.

The dependency of Android developers on OpenSSL may be reasoned with their need to maintain compatibility. However, this dependency leads to some severe vulnerabilities being introduced into Android SSL/TLS stack. Table 5.4 lists the vulnerabilities that are publicly disclosed for each for the OpenSSL versions found in Android devices.

The latest Android vendor device utilizing OpenSSL is Alps, operating on Android 10. Considering that Android 10 was released in 2019, it can be inferred that the OpenSSL version used by Alps is OpenSSL 1.1.1a or a later release [86]. This indicates that the OpenSSL distribution integrated into the Alps network stack was released 6 years prior to the device's release. Furthermore, according the vulnerbaility disclosure dates, most of the vulnerabilities were publicly announced in years 2015 and 2016. This implies that the manufacturing team at Alps had more than three years to update their OpenSSL version. These findings highlight the significance of timely updating OpenSSL distributions to ensure end-user security and maintain vendor reputation.

For example CVE-2016-2182 is a high severity vulnerability that is resulted from improper validation of division results that could result in denial of service attacks on the device running vulnerable versions of OpenSSL. [1] All the vendor devices using OpenSSL as their cryptographic provider, is potentially vulnerable to this threat. As a reference, while OpenSSL shows an alarming number of highly and medium severe vulnerabilities, BoringSSL has only resulted in one public CVE since it's release in 2007 (CVE-2018-12440; CVSS score: 1.9).

[1] CVE-2016-2182. The BN_bn2dec function includes in the crypto/bn/bn_print.c in OpenSSL before releases 1.1.0 does not properly validate the division results, therefore showing potential threat of remote attackers causing denial of service.

OpenSSL version	Vendor	Vulnerabilities (CVEs)
OpenSSL 1.0.1e 11 Feb 2013	Alps Sprd	CVE-2016-2182 CVE-2016-2842 CVE-2016-0705 CVE-2016-6303 CVE-2016-0799 CVE-2016-2177 CVE-2016-6304 CVE-2016-2181 CVE-2016-2183 CVE-2016-2180 CVE-2016-2105 CVE-2016-0798 CVE-2015-3194 CVE-2015-1789 CVE-2016-6302 CVE-2016-0800
OpenSSL 1.0.1j 15 Oct 2014	Alps Oppo Mito Lava	CVE-2016-2182 CVE-2016-6303 CVE-2016-2177 CVE-2016-0799 CVE-2016-2842 CVE-2016-0705 CVE-2015-1789 CVE-2015-3194 CVE-2016-0798 CVE-2016-0797 CVE-2016-2105 CVE-2016-2180 CVE-2016-2183 CVE-2016-6302 CVE-2016-2179 CVE-2016-6304 CVE-2016-6306 CVE-2016-0704 CVE-2016-0800 CVE-2016-2178
OpenSSL 1.0.2e 3 Dec 2015	Alps	CVE-2016-2182 CVE-2016-2177 CVE-2016-0799 CVE-2016-2842 CVE-2016-0705 CVE-2016-6303 CVE-2016-2176 CVE-2016-2105 CVE-2015-3193 CVE-2015-3194 CVE-2016-0797 CVE-2016-2109 CVE-2016-0798 CVE-2016-2106 CVE-2016-2180 CVE-2016-2183 CVE-2016-6302 CVE-2016-2179 CVE-2016-2181 CVE-2016-6304 CVE-2017-3731

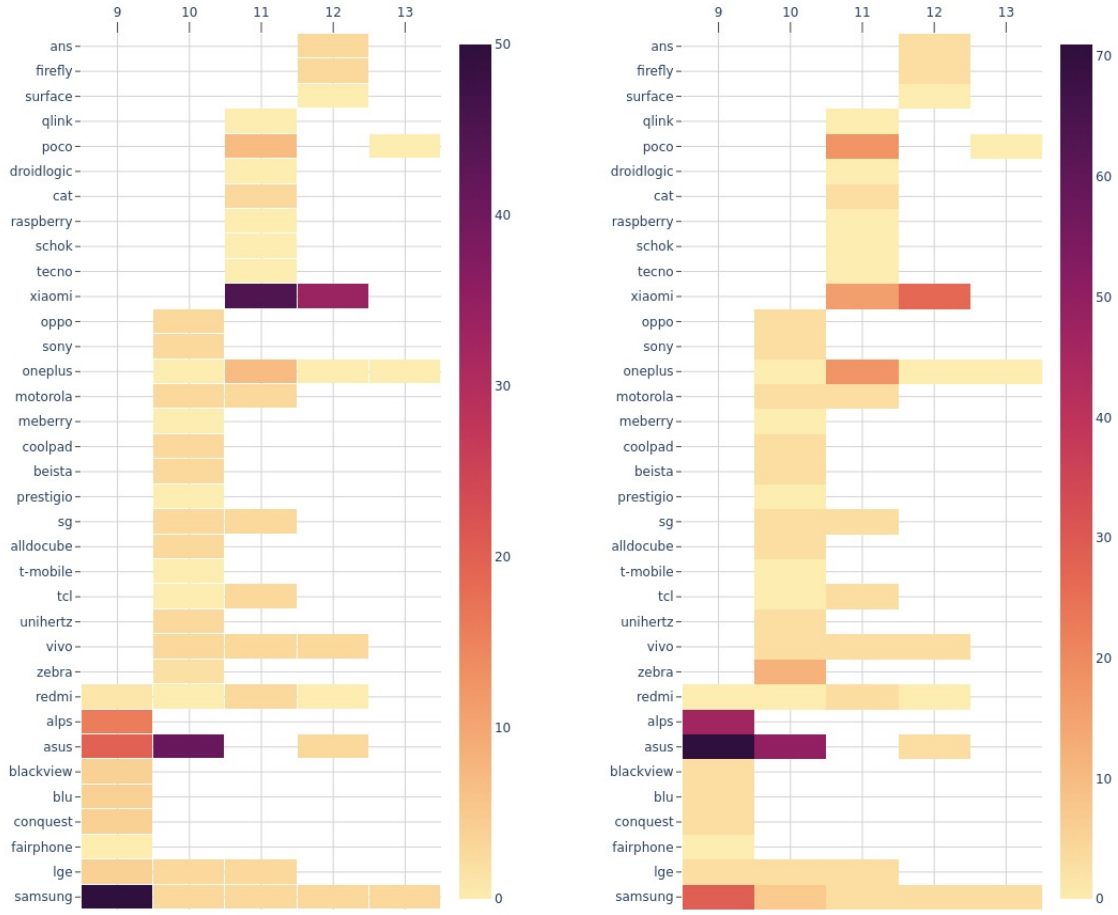
Table 5.4: OpenSSL vulnerabilities disclosed for each OpenSSL version found within the dataset.

5.2.2 Functionality Changes

The differential technique using the function-wise removals and additions shows an significant amount of functionalities being both removed and added. We examine the addition or removal of functionalities in the `libssl.so` object with respect to the

baseline AOSP cryptographic provider.[2]

To infer the purpose of the removed functions, I manually analyze the AOSP source code, and the official documentation of both BoringSSL and OpenSSL. However, my ability to examine the purpose of added functions is limited due to the unavailability of cryptographic providers' source code. As a result, I cannot precisely determine the purpose of added methods and the reason why they exist.



(a) Functions removed from vendors

(b) Functions added by vendors

Figure 5.10: Libssl functionalities removed and added by vendors.

115 unique functions in total were flagged as removed across 25 vendors like Samsung, Zebra, Xiaomi, JGE and Alldocube while the same of set vendors add new functions which are invokable by app developers. Asus devices running on Android 9 shows a high number of additional functionalities, while Alps, Xiaomi and Samsung devices also include a significant number of new functions, but also removed ones

[2] `libssl` is responsible for SSL/TLS protocol implementations is examined against the baseline AOSP cryptographic providers.

for Android version 9 as it can be observed in Figure 5.10: 50 libssl functions are not found in their libssl shared object. Similarly, Xiaomi and Asus devices running on Android 10,11,12 have around 40 removed functionalities. Some of the removed functions, which are listed in Table 5.5, are responsible for critical security aspects of SSL/TLS protocols. Therefore, their absence can alter the intended performance and security guarantees of the protocol, as well as impairing their standard behavior and compatibility.

Xiaomi: Xiaomi version 11 and 12 devices shows that a significant number of functionalities differ when compared to the intended design from the Google AOSP. Specifically, Xiaomi 12 shows that most of the OpenSSL functions adapted by BoringSSL—starting with OPENSSL_lh_—have been replaced with lh_ in their libcrypto implementations. These lh_ functionalities are defined functions in the hash table implementation class implementation by OpenSSL. This example shows how vendors use a combination of OpenSSL and BoringSSL methods in order to achieve and increase the functionalities that they require. Also, the non-usage of EVP_HPKE_ function in Xiaomi version 11 and 12 indicates that the vendor isn't utilizing the HPKE (Hybrid Public Key Encryption) protocol [62]. Although HPKE is known to carry performance overhead above other alternative cryptographic protocols such as the ANSI ECIES (Eliptic Curve Integarted Encryption), in terms of security it strengthens overall security of the SSL/TLS communication by helping to prevent eavesdropping, MITM, and other potential compromises that could lead to decrypting past communications through long-term secret keys. Although EVP_HPKE_ functions are not implemented, Xiaomi has used EVP_HPKE_get_aead to retirve information regarding the HPKE context of the input stream in order retrieve the authenticated encryption with the associated data in Android 12. Xiaomi devices also has not utilized X509_VERIFY_ functionalities that supports the X509 certificate verification process. By removing these supporting functions the target process might not be properly enforce, making the commnication vulnerable.

Samsung: The analysis of Samsung devices running on Android 10, shows that through the vendor modifications additonal cryptographic primitives being called by libssl to be used for protocol implementations. EVP_aead_aes_256_cbc_sha256_tls, EVP_aead_aes_128_cbc_sha256_tls which call upon authenticated encryption using AES256 in CBC(cipher block chaining) with SHA256 message authentication and AES128 in CBC(cipher block chaining) with SHA256 message authentication are widely accepted and approved algorithm due to their strong security properties, therefore these additions might increase the security of the target communication. The addition of EVP_aead_rc4_sha1_tls which is a SHA1 message authentication method using the RC4 steam cipher, which is comparatively old and known to be vulnerbale [61] could be due to support required By legacy systems. Samsung 9 shows an addition of a deprecated class of cryptographic operations including diffie-hellman implementations (detected through the existence of dh_ functions). The deprecation was performed by the Google Android team in the year 2017 [55] but the Samsung devices that were relased in 2018, still includes these cryptographic implementations, this could be due to the requirement of providing support for legacy systems and implementations.

Alps: As we have seen for the JSSE implementation, Alps' cryptographic providers also show significant deviations from the baseline, specially in older Android device running on Android 9 and 10 platform releases. This behavior is apparent if we ana-

lyze the low-level layers of the SSL/TLS stack, as seen on Table 5.6. In this case, the vendor has removed vulnerable and legacy implementations for SSL v3, and replaced them with the recommended [22] EVP_aead_aes_128_gcm_tls13 functionality. The vendor has also incorporated HRSS_ functions which are used to implement post-quantum cryptographic algorithms which are secure against attacks from quantum computers as well as normal computers. Also the added lh_ and sk_functions for hash tables and stacks shows the possibility of developers depending on original OpenSSL methods, similar to Xiaomi. Yet, the removal of critical functions for securing TLS communications at the JSSE level can undermine the security improvements introduced by Alps at the JCA layer.

Removed function	Purpose of the removed functionality	Vendor:Version
SSL_export_early_keying_material	Used to export early mathematical materials that are used for cryptographic operations. This could impact the secure key generation due to not being able to validate the materials being used. This could weaken the overall security of the SSL/TLS connection being established.	Asus:9 Samsung:9 Zebra:10
SSLv3_server_method SSLv3_method SSLv3_client_method	upon examining the SSLv3, this could be an improvement in terms of security. SSL v3 is vulnerable to attacks such as POODLE.	Alps:9 Asus:10 Asus:9
SSL_set_tls13_variant SSL_CTX_set_tls13_variant	sets the TLS 1.3 variant for the SSL/TLS connection. Since TLS 1.3 introduced significant security improvements, the non-existent of this function could weaken the security.	Alps:9 Asus:10 Asus:9
bssl::OpenRecord	Open and process the SSL/TLS records. The inability to decrypt incoming streams and perform security checks could result in insecure information being sent.	Alldocube:10 Alps :9 Asus:10 -12 Beista:10 Blackview:9 Blu:9 Cat:11 Conquest:9 Coolpad:10 Firefly:12 LGE:9 -11 Motorola:10-11 Oneplus:11 Oppo:10 Poco:11 Redmi:11 Samsung:9-13 SG:10-11 Sony:10 TCL:11 Unihertz:10 Vivo:10-12 Xiaomi:11-12

bssl::SealRecordSuffixLen bssl::SealRecordPrefixLen	handle the process of appending additional data to the beginning and the end of the encrypted records before being transmitted. Used to ensure the integrity and authenticity of the data. These include MAC, Padding and Initialization vectors, record length, etc.	Alldocube:10 Alps :9 Asus:10 -12 Beista:10 Blackview:9 Blu:9 Cat:11 Conquest:9 Coolpad:10 Firefly:12 LGE:9 -11 Motorola:10-11 Oneplus:11 Oppo:10 Poco:11 Redmi:11 Samsung:9-13 SG:10-11 Sony:10 TCL:11 Unihertz:10 Vivo:10-12 Xiaomi:11-12
SSL_CTX_set_ed25519_enabled	enables support for Ed25519 signature algorithm, this was added to BoringSSL in 2017 due to being considered superior to DSA.	Asus:9 Oneplus:11 Poco:11 Xiaomi:12
SSL_set_verify_algorithm_prefs	sets the preferred algorithms to be used for signature verification, when removed the lack of prioritization could affect the verification process.	Asus:10 Xiaomi:11
SSL_CTX_set1_ech_keys SSL_ECH_KEYS_add SSL_marshal_ech_config SSL_set1_ech_config_list SSL_set_enable_ech_grease SSL_ECH_KEYS_free SSL_ECH_KEYS_new SSL_ECH_KEYS_has_duplicate_config_id SSL_ECH_KEYS_marshal_retry_configs SSL_ECH_KEYS_up_ref bssl::ssl_is_valid_ech_public_name SSL_get0_ech_retry_configs SSL_ech_accepted SSL_get0_ech_name_override	sets and configures the ability to perform client-side ECH (Encrypted Client Hello), introduced in TLS 1.3 this prevents outsiders from observing the clear-text information about the the connection being established such as the server name indication.	Xiaomi:11 Xiaomi:12
SSL_CTX_set_rsa_pss_rsae_certs_enabled	enables the use of the RSA-PSS signature scheme for the authentication of TLS connections. This provides enhanced security compared to RSA in cases such as collision attacks.	Asus:9 Oneplus:11 Poco:11 Xiaomi:12

Table 5.5: security significant functionalities removed from Android vendors and their functionalities

As shown by the Table 5.5, the unaccessible functionalities in some Android SSL/TLS stacks are critical to Android secure communication. While vendors such as Xiaomi failed to include extensive security functionalities provided by BoringSSL such as ECH and stronger key generation algorithms into their SSL/TLS stack, simple BoringSSL functions that ensures integrity of the messages (sealing records) have been overlooked by 25 vendors.

The above analysis examines the unavailability of critical protocol configuration and implementation functionalities in Android vendors due to the Android supply chain. In order to complement the analysis on vendor protocol implementation, the vendor usage of cryptographic primitive implementations through libcrypto is studied.

Android vendor	Android version	libcrypto functions not used by libssl	libcrypto functions additonally used by libssl
Samsung	9	OPENSSL_realloc EVP_AEAD_CTX_zero CBB_add_asn1_bool CRYPTO_chacha_20 OPENSSL_free CBS_get_asn1_bool CBB_add_asn1_octet_string OPENSSL_malloc EVP_PKEY_set1_RSA EVP_AEAD_CTX_aead EVP_AEAD_CTX_tag_len CRYPTO_tls1_prf X509_STORE_CTX_zero	BN_num_bytes DH_num_bits DH_new BIO_push DH_size DH_compute_key DH_free BIO_s_connect DH_generate_key EVP_aead_rc4_sha1_tls EVP_aead_rc4_sha1_ssl3 DHparams_dup BIO_f_buffer HMAC_CTX_copy_ex ERR_add_error_data RSA_up_ref BN_clear_free CBS_get_last_u8 EVP_PKEY_assign_RSA
Alps	9	EVP_md5 EVP_PKEY_sign ASN1_d2i_bio ASN1_i2d_bio sk_pop_free EVP_PKEY_CTX_free BIO_free_all OPENSSL_cleanse EVP_PKEY_CTX_new EVP_PKEY_verify_init EVP_aead_des_ede3_cbc_sha1_ssl3 EVP_PKEY_sign_init EVP_aead_aes_256_cbc_sha384_tls EVP_aead_aes_128_cbc_sha256_tls EVP_aead_aes_256_cbc_sha256_tls' CRYPTO_chacha_20 EVP_aead_null_sha1_ssl3 EVP_aead_aes_256_cbc_sha1_ssl3 OPENSSL_realloc EVP_aead_aes_128_cbc_sha1_ssl3 EVP_MD_CTX_size EVP_PKEY_verify CBS_stow	HRSS_encap sk_pop_free_ex sk_sort BN_num_bytes HRSS_generate_key EVP_CIPHER_iv_length HRSS_parse_public_key HRSS_marshal_public_key BIO_read_asn1 sk_dup BIO_write_all EVP_aead_aes_128_gcm_tls13 EVP_aead_aes_256_gcm_tls13 lh_retrieve_key HRSS_decap sk_delete
Zebra	10		EVP_MD_type
Samsung	10		EVP_aead_aes_256_cbc_sha256_tls EVP_aead_aes_128_cbc_sha256_tls BIO_push EVP_aead_rc4_sha1_tls BIO_s_connect BIO_f_buffer
Poco	11	OPENSSL_memdup CBS_get_asn1_int64 BIO_get_retry_flags CBB_add_asn1_int64 BIO_set_flags BIO_set_retry_reason EVP_HPKE_get_aead	BIO_copy_next_retry

Xiaomi	11	OPENSSL_memdup CBS_get_asn1_int64 BIO_get_retry_flags CBB_add_asn1_int64 BIO_set_flags BIO_set_retry_reason EVP_HPKE_get_aead OPENSSL_lh_free X509_STORE_CTX_free EVP_MD_CTX_move CBB_add_zeros EVP_HPKE_AEAD_id OPENSSL_lh_retrieve X509_STORE_CTX_get1_chain X509_STORE_CTX_get_error EVP_hpke_aes_128_gcm EVP_HPKE_KEY_copy EVP_HPKE_CTX_setup_recipient EVP_HPKE_CTX_aead X509_STORE_CTX_new EVP_HPKE_CTX_cleanup EVP_HPKE_KEM_id EVP_hpke_hkdf_sha256 EVP_HPKE_KEY_cleanup EVP_HPKE_KEY_public_key EVP_hpke_chacha20_poly1305 OPENSSL_lh_new EVP_HPKE_CTX_seal OPENSSL_lh_num_items EVP_HPKE_CTX_setup_sender EVP_HPKE_CTX_zero EVP_HPKE_CTX_open X509_VERIFY_PARAM_set_hostflags EVP_HPKE_KEY_kem EVP_hpke_x25519_hkdf_sha256 OPENSSL_lh_doall_arg X509_VERIFY_PARAM_set1_host EVP_HPKE_CTX_kdf OPENSSL_lh_delete EVP_HPKE_AEAD_aead OPENSSL_lh_insert EVP_marshal_public_key EVP_HPKE_KEY_zero EVP_HPKE_KDF_id OPENSSL_lh_retrieve_key EVP_hpke_aes_256_gcm	BUF_memdup BIO_copy_next_retry BUF_strdup lh_retrieve X509_STORE_CTX_cleanup lh_free lh_doall_arg lh_num_items lh_insert lh_new lh_delete lh_retrieve_key X509_STORE_CTX_zero

Xiaomi	12	BIO_copy_next_retry OPENSSL_lh_free X509_STORE_CTX_free EVP_MD_CTX_move CBB_add_zeros EVP_HPKE_AEAD_id OPENSSL_lh_retrieve X509_STORE_CTX_get1_chain X509_STORE_CTX_get_error EVP_hpke_aes_128_gcm EVP_HPKE_KEY_copy EVP_HPKE_CTX_setup_recipient EVP_HPKE_CTX_aead X509_STORE_CTX_new EVP_HPKE_CTX_cleanup EVP_HPKE_KEM_id EVP_hpke_hkdf_sha256 EVP_HPKE_KEY_cleanup EVP_HPKE_KEY_public_key EVP_hpke_chacha20_poly1305 OPENSSL_lh_new EVP_HPKE_CTX_seal OPENSSL_lh_num_items EVP_HPKE_CTX_setup_sender EVP_HPKE_CTX_zero EVP_HPKE_CTX_open X509_VERIFY_PARAM_set_hostflags EVP_HPKE_KEY_kem EVP_hpke_x25519_hkdf_sha256 OPENSSL_lh_doall_arg X509_VERIFY_PARAM_set1_host EVP_HPKE_CTX_kdf OPENSSL_lh_delete EVP_HPKE_AEAD_aead OPENSSL_lh_insert EVP_marshal_public_key EVP_HPKE_KEY_zero EVP_HPKE_KDF_id OPENSSL_lh_retrieve_key EVP_hpke_aes_256_gcm	OPENSSL_memdup CBS_get_asn1_int64 BIO_get_retry_flags CBB_add_asn1_int64 BIO_set_flags BIO_set_retry_reason EVP_HPKE_get_aead X509_STORE_CTX_cleanup lh_free lh_doall_arg lh_num_items lh_insert lh_new lh_delete lh_retrieve_key X509_STORE_CTX_zero lh_retrieve

Table 5.6: libcrypto functionalities utlized and non-utilized by libssl.

5.3 JCE customizations

The Java Cryptographic Extension (JCE) serves as a Java wrapper for the native providers within the Android network stack. JCE plays a central role in determining which JCA provider functionalities are accessible to developers during an application or service execution. My focus is on examining the JCE with regard to the provider functionalities that vendors restrict access to, as well as any additional functionalities they may expose to the application layer.

The main analysis of the JCE packages includes the functionality modifications done by the vendors on the default Conscrypt provided by Google through the AOSP. The analysis of Conscrypt's modification shows the set of SSL/TLS functions available for user-space app developers through the JSSE API, as in which JCA functionalities are extended through JCE and available for JSSE APIs. (Section 2.3, Figure 2.2)

5.3.1 Conscrypt Customizations

Figure 5.11 renders the set of Conscrypt functionalities that are both available and unavailable (i.e., likely removed) across devices with regards to their corresponding AOSP baseline. The analysis on Conscrypt classes shows that 6 vendors, namely Alps, Amlogic, Allwinner, yestel, Samsung and Huawei have made removals as well as additions to the default Conscrypt implementation.

However, vendors Alps, Huawei and Allwinner have made significant modifications to the Conscrypt original implementation with an average of 250 functionalities removed. Vendor like Amlogic, Samsung and Yestel are also responsible for a non-negligible number of deviations in both JSSE API and JCA layers if compared to their AOSP equivalent. Even if the JCA provider has been left untouched, if the JCE functionality is missing then app developers will not be able to access it.

(a) Functions removed from vendors

(b) Functions added by vendors

Figure 5.11: Conscrypt functionalities unavailability and availability through vendor SSL/TLS stacks.

As in the case of the JCA analysis, I cannot infer the purpose of these customizations due to Google's Conscrypt documentation being limited on the prupose of each class and function. To overcome this limitation, I manually analyzed the code, allowing me to assess the impact of these customizations. For time considerations, I focus my analysis on the 20 most customized classes using differential code analysis techniques at the smali level. According Google, the team responsible for Conscrypt is selective when choosing the primitives to provide [23]. They focus on the most widespread and secure algorithms. Therefore, it will be important to detect which default Conscrypt operations are not useful from a vendor perspective.

Figure 5.12 shows the distribution of vendor removed Conscrypt methods (in x-axis) in regards to their class (in y-axis). Table 5.7 complements Figure 5.12 and lists some of the most critical functions that have being removed by vendors. However, it is possible that these removals could be due to dependencies on older Conscrypt implementations of Conscrypt, or because they rely on a third-party Conscrypt distribution instead of using Google's official one. As we can see, the most altered class is NativeCrypto class. This class connects the BoringSSL cryptographic implementation to the Java layer of the Android OS. Among the vendor devices being analyzed, 137 native cryptographic functions are missing in Yestel, Samsung, Huawei, Amlogic, Alps and Allwinner vendor implementations. The Conscrypt platform class implemented by Google [3], offers app developers an opportunity to invoke various security-related APIS to customize the default Android platform TLS configuration when opening TLS sockets. For example, the SSLParameterImpl class allows developers to encapsulate all the information on enabled TLS protocols offered by the JCA provider, and the set and order of ciphersuites. Additionally, they also allow performing checks on the operation mode of the SSL socket during connection establishment (Table 5.2, javax.net.SocketFactory - checkOpmode). Another example is the TrustManagerImpl class, which handles certificate chain validations, certificate revocations status, and self-signed root certificates. These also deviate from the baseline AOSP for vendors Huawei, Allwinner and Alps.

[3] Conscrypt platform class - org.conscrypt.Platform

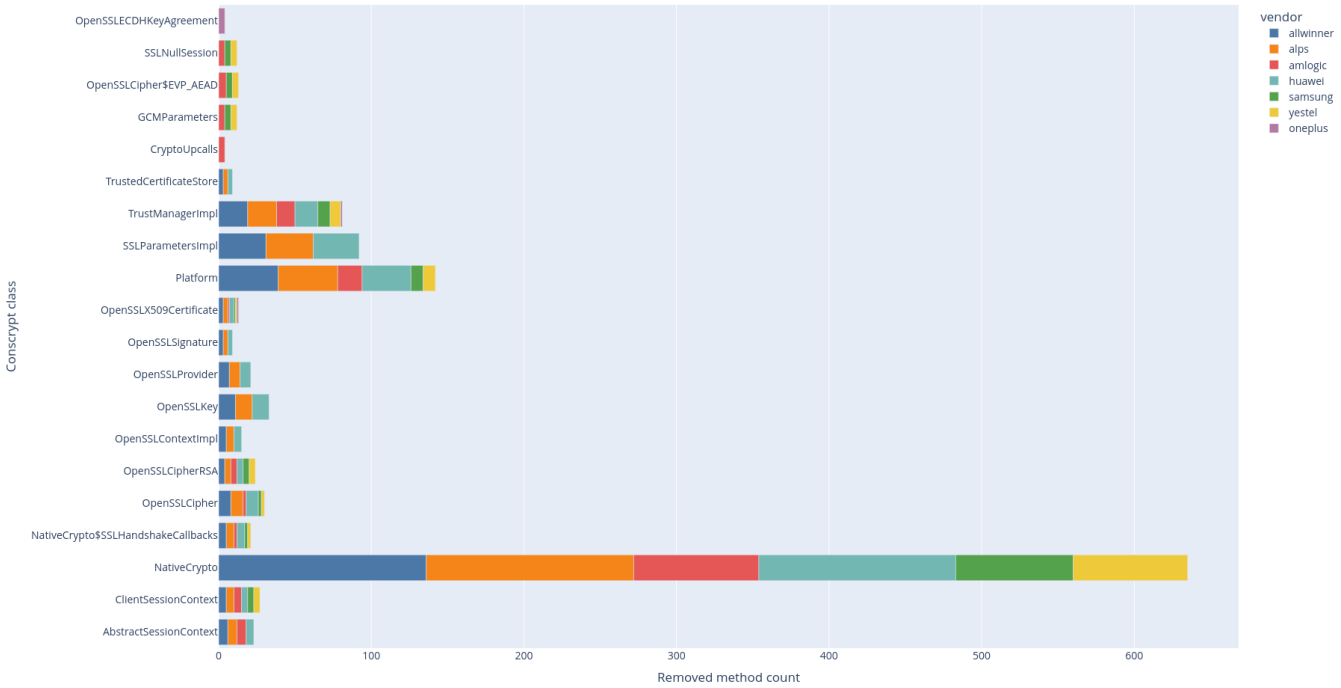


Figure 5.12: Conscrypt classes with the most removed functions compared to the AOSP.

Conscrypt functionality	Impact on perfromance and security	Vendor:Versions
	org.conscrypt.NativeCrypto	
ECDSA_sign ECDSA_size ECDSA_verify	these elliptic curve digital signature algorithm are now deprecated in OpenSSL but not in BoringSSL. This might be an indication that some vendors follow the updates done by OpenSSL compared to BoringSSL due to the limited functionalities.	Samsung:9 Yestel:9 Amlogic:9 Huawei:9 Allwinner:9 Alps:10
EC_KEY_marshal_curve_name EC_KEY_get1_group EC_GROUP_new_arbitrary EC_KEY_parse_curve_name	eclliptic curve-based serialization functions defined by BoringSSL are not defined in these vendor Conscrypt source.	Samsung:9 Yestel:9 Amlogic:9 Huawei:9 Allwinner:9 Alps:10
ENGINE_SSL_do_handshake ENGINE_SSL_force_read ENGINE_SSL_read_BIO_direct ENGINE_SSL_read_BIO_heap ENGINE_SSL_read_direct ENGINE_SSL_shutdown ENGINE_SSL_write_BIO_direct ENGINE_SSL_write_BIO_heap ENGINE_SSL_write_direct	ENGINE_SSL functionalities were added by Google in order to extract information on the TLS connection established and extend the raw JCA implementation. SSL_shutdown function offers secure shutdown which could prevent incomplete connection terminations ultimately leading to an inconsistent state.	Samsung:9 Yestel:9 Amlogic:9-10 Huawei:9 Allwinner:10 Alps:10
EVP_aead_aes_128_gcm EVP_aead_aes_256_gcm	support provided by TLS 1.3 cipher suites is not available for these devices. This behavior was also displayed by the analysis done on JCA providers.	Huawei:9 Allwinner:10 Alps:10
asn1_read_init asn1_read_oid asn1_read_sequence .... (all asn1_ implementations)	these are marked as deprectaed and advised not to be used according to BoringSSL and OpenSSL. The removal could have a positive impact given that these functions are described as "buggy".	Samsung:9 Yestel:9 Amlogic:9-10 Huawei:9 Allwinner:10 Alps:10

setLocalCertsAndPrivateKey	according to the Google source, this extension is used insetting local certificate chains and private key for a SSL/TLS connection. This supports the developers in providing a specific certificate chain that they expect for a certain connection.	Samsung:9 Yestel:9 Amlogic:9-10 Huawei:9 Allwinner:10 Alps:10
org.conscrypt.Platform		
blockGuardOnNetwork	this fuction activate the StrictMode for network operation performed by Conscrypt which is a debugginng functionality that analyzed performance issues that arises from network related tasks.	Huawei:9 Allwinner:10 Alps:10
closeGuardClose closeGuardGet closeGuardOpen closeGuardWarnIfOpen	checks for unclosed streams and marks them as closed or issues a warning in case they are still open. Helps in preventing potential resource leaks.	Huawei:9 Allwinner:10 Alps:10
isCTVerificationRequired	invokes certificate transparency verification, which can be useful when detecting rogue or misused certificates. CT improves security and transparency of certificates issuers by publicly logging certificates.	Samsung:9 Yestel:9 Amlogic:9-10 Huawei:9 Allwinner:10 Alps:10
org.conscrypt.TrustManagerImpl		
checkTrustedRecursive uses: findAllTrustAnchorsByIssuerAndSignature verifyChain sortPotentialAnchors checkBlacklist	Recursively builds a certificate chain until the end result is a valid chain or until all possible paths are exhausted. all the defined methods which are called upon by this method are also unavailable. This could indicate a change in the traditional trust verification process by these vendors.	Huawei:9 Allwinner:10 Alps:10

Table 5.7: Critical conscrypt functions which are unavailable from vendor stacks.

5.3.2 Vendor adaptation of Okhttp and BouncyCastle

The unpacking of the OkHttp and BouncyCastle packages found within the Android devices in the dataset shows that vendors have embedded different releases and derivatives of these third party packages into their SSL/TLS stack. This section of the analysis will examine the exsitence of different releases of the JCE providers in the wild. During the analysis, distribution of OkHttp from the Android ROM developer Xiaomi, showed its existence within all the devices that include their Miui customization layer. These distributions are found within the following vendor (as shown in Table 5.8) in the dataset.

Vendor	Android version
Xiaomi	9-12
Redmi	9-11
Poco	10-12

Table 5.8: miui-okhttp distribution

Another such vendor that modifies the default OkHttp release to fit their requirements in Samsung. This modification includes adding a specific classes to support their Knox security and management framework's VPN implementation. These classes serve the purpose of checking for local proxy ports and managing the proxy credentials.

BouncyCastle which is slowly deprecated by Android, has several releases due to the compatibility and security issues. BouncyCastle has had two main derivatives avail-

able within the Android eco-system; SpongyCastle and StripyCastle as discussed in section 2.3.1.3. The source code level analysis on Bouncycastle files found within the dataset, shows that many devices include the default BouncyCastle distribution, while Zebra android devices running on version 10 showed the inclusion of StripyCastle. According to bouncycastle documentation, StripyCastle which is the FIPS-enabled version of the package supports Android devices upto version 8. The usage of the a deprecated, unmaintained distribution could affect the functionality and security of these devices.

Chapter 6

Conclusion

This dissertation presents the first large-scale study of the impact of device vendor modifications to the Android SSL/TLS stack, and it's implication to end users' and application's network security. The extensibility and openness of the Android ecosystem provides Android device vendors and OEMs the convenience of building Android images that suites their requirements and their hardware. However, the lack of guidelines and controls on such customizations and the whole Android supply chain could introduce security and privacy threats on the devices, and the changes on the Android SSL/TLS stack are no exception.

Utilizing a combination of manual and automated reverse engineering methods—specifically automatic static analysis and code diffing techniques—this study detects and characterizes at scale the customizations that hundreds of device vendors have implemented on three crucial components of the Android SSL/TLS stack. Specifically: (i) the Java secure socket extension API layer, (ii) Java cryptography architecture providers, and (iii) the Java cryptographic extension. In order to obtain a global perspective of vendor customizations on the TLS networking stack, I applied my static analysis pipeline on a dataset of 48,250 different Android images from hundreds of OEMs (dataset overview in Table 4.1). In order to conduct a fair and accurate analysis, I defined a baseline using AOSP versions to detect such vendor customizations automatically.

My findings unveiled a worrying trend of poor vendor practices, as majority of the vendors do not incorporate the latest AOSP changes into their SSL/TLS stack, and that the vendors are slow to incorporate new functionalities and patches. I also found that the vendors are quick to deprecate functionalities, even before the AOSP does. Natively, several vendors use outdated OpenSSL distribution as their default cryptographic provider, instead of BoringSSL which is the default Google-maintained cryptographic provider in the AOSP. In fact, the usage of outdated OpenSSL distributions is a cause for concern as they make devices vulnerable to attacks such as Heartbleed and DROWN. I also find empirical evidence of missing methods that play a critical role in allowing app developers securing TLS flows such as host name verification, certificate validation, and prioritized ciphersuites in the TLS stacks of vendors such as Alps, Sprd, Amlogic.

Overall, my analysis shows that Android device vendors are not consistent in their practices and that there is a significant fragmentation in the Android ecosystem that

not only have an impact on users' security but also on user-space application's needing strong security guarantees. My results call for the urgent need for conducting deeper analysis of the TLS implementations and for tighter controls and guidelines to prevent vendor-induced vulnerabilities.

6.1 Discussion

The issues identified within Android vendor SSL/TLS stacks may have various underlying root causes. One possible reason is that vendors may be using an older platform relase by Google, which lacks the latest best-effort practices added in order strengthen the Android secure communication. Also, some deprecations and removals from the SSL/TLS stack are not reflected in vendor stacks, even after multiple platform releases. This could be due to the compatibility issues vendors might face, as removing these methods from their stacks might break their Android OS functionalities. I also discover common links between vendor supply chains such as partnerships with MNOs and agreements with software interoperatibilty initiative such as OSSii, which impose specific OS structure among the vendirs. Although beneficial to vendors and also to end users, the extent of modifications raises concern.

To address the negative impact of vendor modifications on secure communication capabilities, I propose the following recommendations. It is important to enhance existing certifications and compliance frameworks to specifically address secure communication issues. This can be achieved by expanding their scope to include evaluation criteria for TLS/SSL implementations and network security practices.

Android certified vendors/ODMs Android certified vendors/ODMs are those who have acquired the Play Protect Certification by Google. This certification allows them pre-load Google app suites in their devices and adhere to the CDD and the CTS programs provided by Google (Section 2.2.1). Despite the documentation on CDD and CTS tests providing an overview of the classes that are tested under the unit tests and the compatibility tests for java extension or javax classes, these do not measure Conscrypt compatibility directly through their test cases. Specifically, as identified in my analysis of the JSSE API, JCE and JCA layers, even some vendors which are categorized under the Android certified such as Samsung, Xiaomi, Zebra, Oneplus, Cat, Alldocube, Blu, Oppo and Motorola seem to lack security functionalities that should be part of the vendor network stacks as defined by AOSP. For example, certain Samsung devices lack the support for certificate transparency and validation that certify and revoke server certificates. Similarly, some vendors like Xiaomi do not integrate modern TLS features such as encrypted client hello (ECH) which encrypts all the privacy-sensitive parameters during a TLS handshake from in-path observers. Integrating test suites to validate device's TLS implementation on Android's certification program may be an effective approach to ensure that vendors' implement state-of-the-art security measures on their products, while also preventing platform fragmentation and potential compatibility issues.

Third-party certification The ioXt [42] platform is a global standard for IoT security and privacy that is based on the industry best practices and is developed by a group of industry experts that are applicable for all connected devices [1]. ioXt security and privacy guidelines for android currently enforces the devices manufacturers to

[1] ioXt Android Profile

comply with Google Mobile service certification [32] and FIPS ensured cryptographic implementations. However, these certification platforms currently overlook many of the security vulnerabilities and implementation flaws that I identified. For example, the ioXt certification platform does not enforce the support for certificate transparency and validation that detect mis-issued certificates and revoked certificates. IoxT, nonetheless, offers a good opportunity to enable independent device certification, and for pushing vendors to implement basic and state-of-the-art TLS security mechanisms on their products.

Regulatory compliance On a regional level, establishments such as the EU Cyber Resilience Act (CRA) [26] can be an enabler for Android vendors to obtain aforesaid certifications and to push the industry to comply with the highest security standards. According the CRA documentation, the Android operating system falls under critical elements which requires the involvement of a third-party during the conformity assessment (including the CE marking) and has given an importance to the low-level of cybersecurity which applies to the mobile ecosystem as well. The CRA could prevent the Android vendors from abroad who are manufacturing non-compliant products from entering the extended EU single market, therefore strengthening the Android device security within the EU.

6.2 Future work

This dissertation is the first to provide a comprehensive threat analysis of the vendor-led customization Android SSL/TLS implementations in the wild. The scope presented includes the Google endorsed networking components such as Conscrypt, BoringSSL and Open-sourced component such as the OpenJDK used to implement JSSE APIs. Yet, this dissertation opens a new research direction to analyze in depth the security threats associated with the lack of control over the Android supply chain at lower-layers, and beyond the network stack.

Analysis of third-party componets of the network stack In order to characterize the entire vendor supply chain, differential techniques should be applied to third-party components, OkHttp and BouncyCastle. Although these libraries are not imperative to the functionalities provided by the Android networking stack, they are used to provide additional functionalities and capabilities to the Android developers. A differential analysis where the vendor adaptations of these layers are compared to the Google's adaptations could provide insight into these extended capabilities and their usage within Android handsets in the market.

Vendor usage of OpenSSL as default cryptographic provider A significant finding within this study is the discovery of OpenSSL usage within some vendor Android devices. The definite reason behind this incorporation and the preference over BoringSSL is yet unclear, though it could be a consequence of compatibility requirements that needs to maintained. BoringSSL has specifically narrowed down the original source from OpenSSL to include all the necessary functionalities required by an Android device or any derivative, therefore these findings needs to be further clarified in future research. The outcomes also display that the OpenSSL versions which were cloned during the builds are not the latest version available by the time of the Android releases. When depending on a vulnerability prone source such as the original OpenSSL, it is pivotal that the latest release of the distribution is incorporated.

These restrained actions from the Android manufacturers need to be further analyzed to narrow down the reasoning through future research.

Variation of ciphersuites per OS versions The method-wise diffing conducted on the Java Cryptography Architecture (JCA) providers, presents the possibility to analyze the ciphersuites utilized by different vendors. In response to the increasing security threats in network communication, Google has prioritized the inclusion of only the safest ciphersuites in their cryptographic providers. Monitoring the evolution of ciphersuites used by each vendor's SSL/TLS stacks can provide valuable insights into their ability to adapt to changes in the Android ecosystem. This allows for the classification of vendors as either reliable or unreliable based on the security of the ciphersuites they employ.

Vendor usage of older BoringSSL distributions While the incorporation of BoringSSL is a positive indicator of developer's following best security efforts, there might exist cases where the incorporated source is not the latest distribution released by Google. With the rigorous maintaining done by Google team on BoringSSL, it is vital that the Android derivatives also carry the latest, most secure version of BoringSSL within their networking stacks. The detection of outdated networking fundamentals, can provide insight into the developer best practices and or the lack there of regards to critical SSL.TLS components.

The dissertation provides preliminary findings and methodology for analyzing the customization of critical components by vendors in the Android ecosystem. While this study focuses on critical networking components (specifically on the SSL/TLS stack components), it's important to note that Android's core system components, such as package manager, location manager, and telephony manager, also play a significant role. Conducting a thorough analysis of these core components can shed light on the fragmentation resulting from vendor variations within the supply chains.

The framework presented in this dissertation will be extended during my Ph.D. studies at IMDEA Networks. In fact, this work is being extended for a planned research paper submission for the upcoming Network and Distributed System Security Symposium(NDSS).

Bibliography

[1] Amazon– App store. https://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011. [Accessed 29-May-2023].

[2] Android 6.0 - platform/libcore - Git at Google — android.googlesource.com. https://android.googlesource.com/platform/libcore/+/refs/tags/android-6.0.1_r81/luni/src/main/java/javax/, . [Accessed 04-Jul-2023].

[3] Android 7.0 - platform/libcore - Git at Google — android.googlesource.com. https://android.googlesource.com/platform/libcore/+/refs/tags/android-7.0.0_r1, . [Accessed 04-Jul-2023].

[4] Android 8.1 Features and APIs | Android Developers — developer.android.com. https://developer.android.com/about/versions/oreo/android-8.1, . [Accessed 04-Jul-2023].

[5] Build a Compatible Android Device | Android Open Source Project — source.android.com. https://source.android.com/docs/compatibility, . [Accessed 01-Jun-2023].

[6] Building Android | Android Open Source Project — source.android.com. https://source.android.com/docs/setup/build/building, . [Accessed 17-Jun-2023].

[7] Android – Certified — android.com. https://www.android.com/certified/, . [Accessed 29-May-2023].

[8] Codenames, Tags, and Build Numbers | Android Open Source Project — source.android.com. https://source.android.com/docs/setup/about/build-numbers, . [Accessed 17-Jun-2023].

[9] Configuring ART | Android Open Source Project — source.android.com. https://source.android.com/docs/core/runtime/configure, . [Accessed 02-Jun-2023].

[10] java.net | Android Developers — developer.android.com. https://developer.android.com/reference/java/net/package-summary, . [Accessed 02-Jun-2023].

[11] java.security | Android Developers — developer.android.com. https://developer.android.com/reference/java/security/package-summary, . [Accessed 02-Jun-2023].

[12] javax.net | Android Developers — developer.android.com. https://

developer.android.com/reference/javax/net/package-summary, . [Accessed 02-Jun-2023].

[13] javax.net.ssl | Android Developers — developer.android.com. https://developer.android.com/reference/javax/net/ssl/package-summary, . [Accessed 02-Jun-2023].

[14] Android Open Source Project — source.android.com. https://source.android.com/, . [Accessed 29-May-2023].

[15] Platform architecture | Android Developers — developer.android.com. https://developer.android.com/guide/platform, . [Accessed 30-Jun-2023].

[16] SDK Platform Tools release notes | Android Studio | Android Developers — developer.android.com. https://developer.android.com/tools/releases/platform-tools, . [Accessed 01-Jun-2023].

[17] Apache HttpClient Overview — hc.apache.org. https://hc.apache.org/httpcomponents-client-5.2.x/. [Accessed 04-Jul-2023].

[18] Baidu– App store. https://shouji.baidu.com/. [Accessed 29-May-2023].

[19] GitHub - bcgit/bc-java: Bouncy Castle Java Distribution (Mirror) — github.com. https://github.com/bcgit/bc-java. [Accessed 29-Jun-2023].

[20] platform/bionic - Git at Google — android.googlesource.com. https://android.googlesource.com/platform/bionic/. [Accessed 01-Jun-2023].

[21] BoringSSL - Headers — commondatastorage.googleapis.com. https://commondatastorage.googleapis.com/chromium-boringssl-docs/headers.html. [Accessed 04-Jun-2023].

[22] BoringSSL - cipher.h — commondatastorage.googleapis.com. https://commondatastorage.googleapis.com/chromium-boringssl-docs/cipher.h.html. [Accessed 28-Jun-2023].

[23] Conscrypt Capabilities — android.googlesource.com. https://android.googlesource.com/platform/external/conscrypt. [Accessed 04-Jun-2023].

[24] CVE-2018-12440 : BoringSSL through 2018-06-14 allows a memory-cache side-channel attack on DSA signatures, aka the Return Of the Hidden N — cvedetails.com. https://www.cvedetails.com/cve/CVE-2018-12440/, . [Accessed 05-Jun-2023].

[25] Gnupg Libgcrypt : CVE security vulnerabilities, versions and detailed reports — cvedetails.com. https://www.cvedetails.com/product/25777/Gnupg-Libgcrypt.html?vendor_id=4711, . [Accessed 05-Jul-2023].

[26] Cyber Resilience Act — digital-strategy.ec.europa.eu. https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act. [Accessed 02-Jul-2023].

[27] openssl/README-FIPS.md at master · openssl/openssl — github.com. https://github.com/openssl/openssl/blob/master/README-FIPS.md. [Accessed 06-Jun-2023].

[28] GitHub - fesh0r/fernflower: Unofficial mirror of FernFlower Java decompiler (All pulls should be submitted upstream) — github.com. https://github.com/fesh0r/fernflower, . [Accessed 10-Jun-2023].

[29] GitHub - google/smali — github.com. https://github.com/google/smali, . [Accessed 10-Jun-2023].

[30] GitHub - pxb1988/dex2jar: Tools to work with android .dex and java .class files — github.com. https://github.com/pxb1988/dex2jar, . [Accessed 10-Jun-2023].

[31] GitHub - testwhat/SmaliEx: A wrapper to get de-optimized dex from odex/oat/vdex. — github.com. https://github.com/testwhat/SmaliEx, . [Accessed 10-Jun-2023].

[32] Android – Google Mobile Services — android.com. https://www.android.com/gms/. [Accessed 02-Jul-2023].

[33] The GNU Privacy Guard — gnupg.org. https://gnupg.org/index.html, . [Accessed 06-Jun-2023].

[34] GnuTLS — gnutls.org. https://www.gnutls.org/, . [Accessed 06-Jun-2023].

[35] boringssl - Git at Google — boringssl.googlesource.com. https://boringssl.googlesource.com/boringssl, . [Accessed 04-Jun-2023].

[36] Firmware Scanner - Apps on Google Play — play.google.com. https://play.google.com/store/apps/details?id=org.imdea.networks.iag.preinstalleduploader&hl=en&gl=US, . [Accessed 09-Jun-2023].

[37] Google Pixel Phones — store.google.com. https://store.google.com/category/phones?hl=es, . [Accessed 01-Jun-2023].

[38] Volley overview — google.github.io. https://google.github.io/volley/, . [Accessed 04-Jul-2023].

[39] refs/heads/master - platform/libcore - Git at Google — android.googlesource.com. https://android.googlesource.com/platform/libcore/+/refs/heads/master, . [Accessed 01-Jun-2023].

[40] Porting from OpenSSL to BoringSSL — boringssl.googlesource.com. https://boringssl.googlesource.com/boringssl/+/HEAD/PORTING.md, . [Accessed 05-Jul-2023].

[41] Java Obfuscator and Android App Optimizer | ProGuard — guardsquare.com. https://www.guardsquare.com/proguard. [Accessed 10-Jun-2023].

[42] ioXt - The Global Standard for IoT Security — ioxtalliance.org. https://www.ioxtalliance.org/. [Accessed 02-Jul-2023].

[43] LibreSSL — libressl.org. https://www.libressl.org. [Accessed 06-Jun-2023].

[44] nm(1) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man1/nm.1.html, . [Accessed 04-Jul-2023].

[45] objdump(1) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man1/objdump.1.html, . [Accessed 04-Jul-2023].

[46] Cryptographic Module Validation Program | CSRC | CSRC — csrc.nist.gov. https://csrc.nist.gov/projects/cryptographic-module-validation-program/certificate/3753, . [Accessed 04-Jun-2023].

[47] NVD - cve-2014-0160 — nvd.nist.gov. https://nvd.nist.gov/vuln/detail/cve-2014-0160, . [Accessed 05-Jul-2023].

[48] OpenJDK — openjdk.org. https://openjdk.org, . [Accessed 29-May-2023].

[49] platform/external/openssl - Git at Google — android.googlesource.com. https://android.googlesource.com/platform/external/openssl, . [Accessed 04-Jun-2023].

[50] SSLContext. https://docs.oracle.com/javase/9/docs/api/javax/net/ssl/SSLContext.html, . [Accessed 04-Jul-2023].

[51] SSLParameters setciphersuites. https://docs.oracle.com/en/java/javase/20/docs/api/java.base/javax/net/ssl/SSLParameters.html#setCipherSuites(java.lang.String%5B%5D), . [Accessed 04-Jul-2023].

[52] SSLParameters setservernames. https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLParameters.html#setServerNames-java.util.List-, . [Accessed 04-Jul-2023].

[53] SSLParameters setsnimatchers. https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLParameters.html#setSNIMatchers-java.util.Collection-, . [Accessed 04-Jul-2023].

[54] Botan: Crypto and TLS for Modern C++. https://botan.randombit.net. [Accessed 06-Jun-2023].

[55] boringssl.googlesource.com. https://boringssl.googlesource.com/boringssl.git/+/7e06de5d2d1b53c57c0c81e8d6ba4122b64cf626. [Accessed 28-Jun-2023].

[56] git-repo - Git at Google — gerrit.googlesource.com. https://gerrit.googlesource.com/git-repo. [Accessed 17-Jun-2023].

[57] GitHub - rtyley/spongycastle: Spongy Castle - a repackage of Bouncy Castle for Android (which ships a crippled version of BC) — github.com. https://github.com/rtyley/spongycastle. [Accessed 29-Jun-2023].

[58] Retrofit — square.github.io. https://square.github.io/retrofit/. [Accessed 04-Jul-2023].

[59] ssdeep - Fuzzy hashing program — ssdeep-project.github.io. https://ssdeep-project.github.io/ssdeep/index.html. [Accessed 19-Jun-2023].

[60] Android Dumps · GitLab — dumps.tadiphone.dev. https://dumps.tadiphone.dev/dumps. [Accessed 10-Jun-2023].

[61] Nadhem J AlFardan, Daniel J Bernstein, Kenneth G Paterson, Bertram Poettering, and Jacob CN Schuldt. On the security of rc4 in tls. In *USENIX Security Symposium*, volume 2013, 2013.

[62] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. RFC 9180, February 2022. URL https://www.rfc-editor.org/info/rfc9180.

[63] Eduardo Blázquez, Sergio Pastrana, Álvaro Feal, Julien Gamba, Platon Kotzias, Narseo Vallina-Rodriguez, and Juan Tapiador. Trouble over-the-air: An analysis of fota apps in the android ecosystem. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1606–1622. IEEE, 2021.

[64] Jenny Blessing, Michael A Specter, and Daniel J Weitzner. You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries. *arXiv preprint arXiv:2107.04940*, 2021.

[65] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84, 2013.

[66] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Firmscope: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2379–2396, 2020.

[67] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61, 2012.

[68] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136. IEEE, 2017.

[69] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1039–1055. IEEE, 2020.

[70] Julien Gamba, Álvaro Feal, Eduardo Blazquez, Vinuri Bandara, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. Mules and permission laundering in android: Dissecting custom permissions in the wild. *IEEE Transactions on Dependable and Secure Computing*, pages 1–18, 2023. doi: 10.1109/TDSC.2023.3288981.

[71] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings 5*, pages 291–307. Springer, 2012.

[72] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, volume 14, page 19, 2012.

[73] David Sounthiraraj Justin Sahs Garret Greenwood and Zhiqiang Lin Latifur Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Network and Distributed System Security Symposium (NDSS). Internet Society, San Diego, CA*, pages 1–14, 2014.

[74] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Xiaofeng Liu, Song Liu, Lingyun Ying, Shanqing Guo, Yuanzhi Li, Meining Nie, and Haixin Duan. Large-scale security measurements on the android firmware ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1257–1268, 2022.

[75] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. Predictability of android openssl's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 659–668, 2013.

[76] Jozef Kostelanský and Ľubomír Dedera. An evaluation of output from current java bytecode decompilers: Is it android which is responsible for such quality boost? In *2017 Communication and Information Technologies (KIT)*, pages 1–6, 2017. doi: 10.23919/KIT.2017.8109451.

[77] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of age: A longitudinal study of tls deployment. In *Proceedings of the Internet Measurement Conference 2018*, pages 415–428, 2018.

[78] Jaeho Lee and Dan S Wallach. Removing secrets from android's tls. In *NDSS*, 2018.

[79] Douglas J Leith. Mobile handset privacy: Measuring the data ios and android send to apple and google. In *Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part II 17*, pages 231–251. Springer, 2021.

[80] Douglas J Leith. What data do the google dialer and messages apps on android send to google? In *Security and Privacy in Communication Networks: 18th EAI International Conference, SecureComm 2022, Virtual Event, October 2022, Proceedings*, pages 549–568. Springer, 2023.

[81] Haoyu Liu, Paul Patras, and Douglas J Leith. Android mobile os snooping by samsung, xiaomi, huawei and realme handsets. *Trinity College Dublin, Tech. Report*, 2021.

[82] Zane Ma, James Austgen, Joshua Mason, Zakir Durumeric, and Michael Bailey. Tracing your roots: exploring the tls trust anchor ecosystem. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 179–194, 2021.

[83] Lucky Onwuzurike and Emiliano De Cristofaro. Danger is my middle name: experimenting with ssl vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 1–6, 2015.

[84] Inc. OpenSSL Foundation. Guide to DROWN - OpenSSL Blog — openssl.org. `https://www.openssl.org/blog/blog/2016/03/01/an-openssl-users-guide-to-drown/`, . [Accessed 05-Jul-2023].

[85] Inc. OpenSSL Foundation. /index.html — openssl.org. https://www.openssl.org, . [Accessed 29-May-2023].

[86] Inc. OpenSSL Foundation. /news/openssl-1.1.1-notes.html — openssl.org. https://www.openssl.org/news/openssl-1.1.1-notes.html, . [Accessed 05-Jul-2023].

[87] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 87–102. IEEE, 2021.

[88] Amogh Pradeep, Muhammad Talha Paracha, Protick Bhowmick, Ali Davanian, Abbas Razaghpanah, Taejoong Chung, Martina Lindorfer, Narseo Vallina-Rodriguez, Dave Levin, and David Choffnes. A comparative analysis of certificate pinning in android & ios. In *Proceedings of the 22nd ACM Internet Measurement Conference*, pages 605–618, 2022.

[89] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. Studying tls usage in android apps. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 350–362, 2017.

[90] Inc. Square. Overview - OkHttp — square.github.io. https://square.github.io/okhttp. [Accessed 29-May-2023].

[91] Inc. http://www.synopsys.com/ Synopsys. Heartbleed Bug — heartbleed.com. https://heartbleed.com. [Accessed 04-Jun-2023].

[92] Vasant Tendulkar and William Enck. An application package configuration approach to mitigating android ssl vulnerabilities. *arXiv preprint arXiv:1410.7745*, 2014.

[93] Narseo Vallina-Rodriguez, Johanna Amann, Christian Kreibich, Nicholas Weaver, and Vern Paxson. A tangled mass: The android root certificate stores. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 141–148, 2014.

[94] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *Proceedings of the Internet Measurement Conference 2018*, pages 293–307, 2018.

[95] wolfSSL. wolfSSL Embedded SSL/TLS Library. https://www.wolfssl.com/products/wolfssl/. [Accessed 06-Jun-2023].

[96] Daoyuan Wu, Debin Gao, Rocky KC Chang, En He, Eric KT Cheng, and Robert H Deng. Understanding open ports in android applications: Discovery, diagnosis, and security assessment. 2019.

[97] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634, 2013.

[98] Min Zheng, Mingshen Sun, and John CS Lui. Droidray: a security evaluation system for customized android firmwares. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 471–482, 2014.

[99] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423. IEEE, 2014.