Beneath the Surface: An Analysis of OEM Customizations on the Android TLS Protocol Stack

Vinuri Bandara[†‡], Stijn Pletinckx[§], Ilya Grishchenko[§], Christopher Kruegel[§], Giovanni Vigna[§], Juan Tapiador[‡], Narseo Vallina-Rodriguez[†]
[†]*IMDEA Networks Institute, Madrid, Spain* [‡]*Universidad Carlos III de Madrid, Spain*
[§]*University of California, Santa Barbara, CA, USA*
{*vinuri.bandara,narseo.vallina*}*@imdea.org,*
{*stijn,grishchenko,vigna*}*@ucsb.edu, chris@cs.ucsb.edu,*
jestevez@inf.uc3m.es

***Abstract*—The open-source nature of the Android Open Source Project (AOSP) allows Original Equipment Manufacturers (OEMs) to customize the Android operating system, contributing to what is known as *Android fragmentation*. Google has implemented the Compatibility Definition Document (CDD) and the Compatibility Test Suite (CTS) to ensure the integrity and security of the Android ecosystem. However, the effectiveness of these policies and measures in warranting OEM compliance remains uncertain. This paper empirically studies for the first time the nature of OEM customizations in the Android TLS protocol stack, and their security implications on user-installed mobile apps across thousands of Android models. We find that approximately 80% of the analyzed Android models deviate from the standard AOSP TLS codebase and that OEM customizations often involve code changes in functions used by app developers for enhancing TLS security like end-point and certificate verification. Our analysis suggests that these customizations are likely influenced by factors such as manufacturers' supply chain dynamics and patching prioritization tactics, including the need to support legacy components. We conclude by identifying potential root causes and emphasizing the need for stricter policy enforcement, better supply chain controls, and improved patching processes across the ecosystem.**

1. Introduction

The Android Open Source Project (AOSP) allows Original Equipment Manufacturers (OEMs) to introduce proprietary features in their devices. However, extending or modifying the original AOSP codebase has two collateral implications: the phenomenon known as *Android fragmentation* [49, 95] and the risk of introducing security vulnerabilities. Google has implemented the Compatibility Definition Document (CDD) [22] and the Compatibility Test Suite (CTS) [85] to ensure that OEMs maintain API integrity and permission model consistency, and enhance the security of the Android ecosystem. CDD policies define security and compatibility requirements [75], while CTS is a suite of automated tests to ensure devices adhere to CDD specifications, including signatures, platform APIs, and permissions checks. Device models passing the CTS tests can be sold as "Android-compliant."

While these measures show Google's attempts to harmonize the Android ecosystem [98], researchers have reported pitfalls in their application [84]. However, prior research has ignored the security risks introduced by OEM customizations to the Android TLS protocol stack, which manages TLS-based traffic and cryptographic functions. Android's TLS stack is composed of four components: the Java Secure Socket Extension (JSSE), the Java Cryptographic Extension (JCE), the Java Cryptographic Architecture (JCA) providers, and the trusted Certificate Authority (CA) root store. These components have undergone continuous security improvements to mitigate vulnerabilities. For example, in 2014, AOSP switched its JCA provider from OpenSSL to BoringSSL in response to the Heartbleed bug [29]. Similarly, Android has supported the most recent TLS standard, TLS 1.3, since Android 10 [4]. However, not all OEMs will diligently follow upstream updates. In fact, vendors may modify or remove functions from the TLS stack for reasons such as maintaining compatibility with legacy systems. These deviations can result in API functional inconsistencies across device models, even for the same version release. This not only increases platform fragmentation but also exposes user-space apps to multiple network threats, including in/on-path intercepting proxies and surveillance if unaware app developers do not handle API inconsistencies correctly [73, 78, 86, 94].

This paper presents the first large-scale analysis of OEM modifications to the Android TLS stack across thousands of Android models. Our research seeks answers to the following three research questions:

- **RQ1** What customizations do OEMs apply to the Android TLS stack (JSSE, JCE, JCA, and trusted CA root store), and what are their security implications?
- **RQ2** Do OEM TLS customizations impact on user-space Android apps' functioning and security?
- **RQ3** Are Google's CDD and CTS effective at ensuring TLS API consistency and forcing OEMs to diligently follow AOSP upstream updates?

To answer these questions, we developed a methodology that leverages code diffing techniques to compare a large amount of OEM codebases with the corresponding baseline AOSP codebase. Our dataset, collected through crowd-sourcing, includes 53,189 Android images from 370 OEMs, comprising 16,789 models running Android versions 9 through 15. This represents the largest and most diverse empirical analysis of critical Android subsystems and OEM customizations to date. The key contributions

and findings of our study are:

1) *Widespread OEM customizations.* Approximately 80% of OEM models deviate from the AOSP codebase. The most critical customizations occur in the lower layers (JCE, JCA) and the CA root store, many of which have clear platform fragmentation and security implications. We classify the security impact of these customizations into seven categories. Under this classification, we find examples of modifications that prevent app developers from effectively blocking clear-text traffic, disabling deprecated standards vulnerable to known attacks such as POODLE [46], or modifying functionality related to certificate blocklisting and pinning. We also observe devices lacking critical updates for modern cryptographic implementations. Additionally, 274 OEMs either add or remove certificates to/from the AOSP trusted root CA store, including untrusted CAs like Comodo and TrustCor (§5). To systematically assess these risks, we define a threat model (§3) that categorizes adversaries and attack scenarios, providing a structured framework to evaluate the security implications of TLS customizations.
2) *Impact on user-space apps.* OEM customizations may negatively impact app developers' efforts to secure network communications on their apps, leading to varying and unexpected security properties across models. We demonstrate the real-world impact of TLS customizations by studying the dependencies that 20k Android apps available on the Google Play Store have on methods removed or tampered with by OEMs. Over 75% of the analyzed apps use at least one of the removed functions. We report evidence of app developers wrongly and correctly handling API inconsistencies in their apps, including popular games like Temple Run 2—with over 1 billion installations—and popular third-party advertising SDKs like Mintegral, which are embedded in hundreds of thousands of apps (§6).
3) *Inconsistencies in OEM patching and effectiveness of CDD/CTS.* Most OEMs show significant discrepancies in maintaining consistency with AOSP, with popular models like the Samsung Galaxy S20 Ultra missing updates and dependencies. This highlights the varied and inconsistent approaches followed by OEMs in keeping up with AOSP security patches, which overall contributes to Android fragmentation. We discuss how Google's current certification efforts may be insufficient to prevent issues regarding API inconsistency, adherence to upstream changes, root store management, and other compliance requirements by measuring the prevalence of certain violations, even in Google-certified models. This highlights the need for a broader scope in Google's CTS tests for catching security and compliance issues more effectively (§7).

Our results indicate the need for proactive and holistic measures such as strict testing frameworks to harmonize Android TLS API security guarantees and ensure compliance with security standards and best practices. As regulations like the EU Cyber Resilience Act (CRA) emerge, regulatory bodies have the opportunity to establish clearer standards that could enhance compliance checks(§7.3).

Responsible Disclosure. We shared a preliminary version of this paper with Google in late 2024. We also filed a bug report to Google and MIUI for one of our findings that is under investigation and marked as a security issue (see details in §5). The paper's dataset and code artifacts are available at https://github.com/OEM-customization/. Appendix A details our open-science approach.

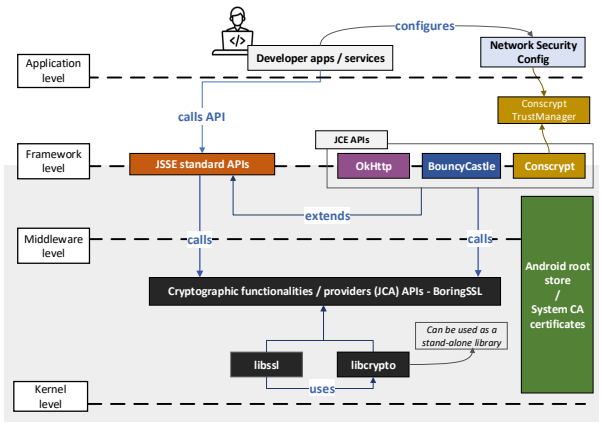

Figure 1: Architecture of the Android TLS stack.

2. Background

Android's networking and cryptographic functions are implemented across several layers, written in both Java and native code, as shown in Figure 1. This multi-layered architecture enables app developers to easily open secure connections with remote endpoints by invoking the Java Secure Socket Extension (JSSE) APIs (§2.1), which interact with (i) the Java Cryptographic Extension (JCE) (§2.3) for cryptographic operations like encryption, decryption, and key management, and (ii) the Java Cryptographic Architecture (JCA) (§2.2) that provides a foundational platform for implementing encryption algorithms and managing cryptographic keys. While JCE has now been integrated into the broader JCA framework [17], in this paper we analyze JCE separately from JCA for clarity and structural purposes. Finally, (iii) the trusted root CA store (§2.4) acts as a trust anchor for the Public Key Infrastructure (PKI), helping to validate the authenticity of server X.509 certificates during the TLS handshake.

2.1. Java Secure Socket Extension (JSSE)

Android app developers can use JSSE APIs to open and manage secure TLS/SSL connections. JSSE is a Java-based framework and set of APIs implementing functions to open Java TLS sockets, modify the app's Network Security Configuration file to trust self-signed X.509 certificates, implement custom hostname verification using the HostnameVerifier interface with classes (e.g., using SSLSocket), or define a denylist of untrusted or compromised CAs, like Comodo and DigiNotar, which may still be present in the handsets' trusted root CA store (§2.4).

From an architectural perspective, JSSE bridges user-space apps with the JCA (§2.2) and the underlying JCE (§2.3) security providers, implemented in both Java and native code. The AOSP sources its JSSE classes from OpenJDK [81]. However, OEMs have the flexibility to import a version of OpenJDK of their choice into the

TABLE 1: Key security threats arising from OEM TLS customizations (NA: Network adversaries, MA: Malicious apps).

Code	Threat Category	Security Impact	Attack Vector
TN1	Outdated TLS components	Increases the attack surface by exposing devices to vulnerabilities already patched in newer versions.	NA forces outdated protocol negotiation (e.g., SSLv3, TLS 1.0) to weaken encryption and enable downgrade attacks like POODLE, DROWN, and BEAST.
TN2	Weak or missing encryption	Exposes sensitive data by allowing unencrypted communication or weak ciphers.	NA manipulates fallback mechanisms to downgrade secure connections, intercepting login credentials or session tokens.
TN3	Insecure endpoint validation	Weak hostname verification or improperly trusted certificates allow adversaries to impersonate trusted entities.	NA uses DNS spoofing and rogue certificates to redirect users to malicious servers, enabling phishing and person-in-the-middle attacks.
TA1	App-level exploits	Bypasses key security mechanisms like certificate pinning, allowing unauthorized access or data interception.	MA exploits (intentionally triggers) customizations to manipulate network traffic, tricking users into accepting invalid certificates and enabling person-in-the-middle attacks.

path `/libcore`—Google's implementation of some core Java libraries that also provide POSIX system calls from Java—as described in AOSP's documentation [37]. This flexibility allows OEMs to modify JSSE class signatures and implementations, potentially contributing to platform API fragmentation and impacting the security guarantees of apps relying on standard Android TLS APIs.

2.2. Java Cryptographic Architecture (JCA)

The JCA, also known as the Cryptographic Service Provider (CSP), is a rich framework implemented in native and Java code. These components provide the underlying cryptographic primitives and protocols for the JSSE (§2.1) and JCE (§2.3), essential for TLS communications, digital signatures, and key management. In AOSP, BoringSSL serves as the default JCA provider [23].

BoringSSL, developed by Google as a derivative of OpenSSL in response to the 2014 Heartbleed vulnerability [29], replaced OpenSSL starting with Android 7 in 2016. BoringSSL, similar to OpenSSL, compiles into two separate shared objects, `libssl` and `libcrypto`, which handle secure protocol implementations and cryptographic primitives. `libcrypto` also serves as the provider of cryptographic primitives for `libssl`. OEMs can modify JCA by replacing or supplementing BoringSSL with alternative open-source providers such as OpenSSL [55] and Libgcrypt [28], or use proprietary solutions such as Samsung Knox [1]. While this flexibility allows OEMs to enhance the cryptographic capabilities of their products, Google advises against practices such as including deprecated algorithms or insecure providers [15].

2.3. Java Cryptographic Extension (JCE)

The JCE is an integral part of the broader JCA, which contains key cryptographic classes responsible for encryption, decryption, and key management operations. AOSP's JCE is formed by several key components:

- **Conscrypt**, introduced in Android 11, implements key cryptographic and TLS support functions. As a Java Security Provider (JSP), it leverages BoringSSL to perform cryptographic operations like encryption and secure communication using TLS. The `libjavacrypto.so` library bridges BoringSSL and the Java components of the Android TLS stack, enabling smooth cryptographic operations through the Java Native Interface (JNI). Conscrypt's `SSLEngine` directly supports `SSLSocket`, creating a unified cryptographic framework.
- **OkHttp** is the principal Java-based HTTP(S) library, which works with Conscrypt to ensure encrypted data transmission that adheres to the latest TLS standards. OkHttp optimizes network performance through features like connection pooling and caching, while ensuring secure data transmission. Although OkHttp is tightly integrated with Android's cryptographic framework and is the default HTTP(S) library, OEMs can replace it with alternatives such as Retrofit [32] or Volley [33].
- **BouncyCastle** historically implemented a wide range of cryptographic algorithms. Since Android 12, many of its deprecated implementations (including all AES algorithms) have been removed and replaced by Conscrypt's functions [77]. Yet, OEMs can still include Bouncy Castle in their products if needed for legacy support. Google advises against this practice, as it may expose devices to outdated cryptographic standards [6].

2.4. The Trusted Root CA Store

The Public Key Infrastructure (PKI) is an essential framework for procedures and policies governing public-key encryption. The Android trusted root Certificate Authorities (CA) store contains trusted root certificates required to verify the identity of remote endpoints before establishing secure TLS connections. AOSP includes 129-138 vetted root certificates (Android 9-14), issued by CAs audited for compliance with industry standards and security practice when issuing certificates [92]. If a root certificate's trust is compromised, Google removes it from the trusted root store [93, 96]. Android's root store overlaps with Chromium's to ensure cross-platform consistency and maintain a reliable trust model.

OEMs can add or remove CAs from the trusted root CA store, as prior research shows [94]. However, Google advises caution due to potential security risks [61]. Privileged components, such as Firmware-over-the-Air (FOTA) providers, can dynamically remove deprecated or expired X.509 certificates from the root store, but also can introduce rogue CAs [69]. The lack of centralized oversight over root store modifications increases the risk of malicious software gaining privileged access [42]. Since all root CAs are equally trusted by default, Android provides APIs to denylist untrusted certificates or CAs, mitigating threats from compromised CAs [10].

3. Threat Model

The security of the Android TLS stack is built on the core components described in §2. OEMs introduce

security risks when they modify these TLS components, deviating from Google's implementation, for example by modifying cryptographic configurations, the trusted root certificate store or overriding default network APIs. To systematically assess these risks, we define a threat model that provides a structured framework for evaluating the security impact of OEM modifications, guiding our empirical analysis. We categorize adversaries capable of exploiting these vulnerabilities into:

- **Network adversaries (NA)**: Attackers who intercept, manipulate, or downgrade encrypted communications over untrusted networks or on-path attacks, exploiting weakened encryption or legacy TLS versions.
- **Malicious apps (MA)**: Apps that exploit misconfigured TLS, improperly manage trust stores, or expose system APIs to disable security mechanisms and gain unauthorized access to sensitive data.

Using the methodology described in §4, we extract and analyze TLS components to identify OEM-introduced customizations. In §5, we map our empirical findings to the defined threat model where relevant, assessing their security implications. This structured approach provides a clear understanding of how TLS customizations contribute to security risks in the Android ecosystem.

4. Methodology

To answer our research questions, we built an analysis pipeline (see Figure 2) based on differential code analysis techniques (diffing) to automatically identify deviations in OEM's codebase from their AOSP equivalent. Our pipeline consists of three stages: (i) OEM image collection (§4.1); (ii) identification of the corresponding AOSP baseline implementation (§4.2); and (iii) detection of the OEM customizations with respect to the corresponding baseline using diffing techniques (§4.3).

4.1. OEM Image Collection

We use two complementary crowd-sourced datasets to gather a representative dataset of Android images from various OEMs, models, and OS versions:

- FirmwareScanner [21] is a tool we developed and published on Google Play to crowdsource Android image collection. It scans the `system/`, `vendor/`, `ODM/`, and `product/` partitions of volunteers' devices to gather device-specific metadata, preinstalled apps (.apk files), X.509 root CA certificates, and native libraries in an ethical and privacy-preserving manner (see ethics discussion below) [59]. Our analysis uses 53,189 Android images (versions 9-15) collected using this tool. Rooted devices are excluded to reduce analysis bias.
- Android Dumps [20] is a free web platform that allows users to upload Android firmware images to a shared repository. It complements FirmwareScanner with 1,925 images for Android 12-14.

Component extraction. We manually inspected the AOSP code and the crowd-sourced Android images to locate the paths of compiled networking components across OEMs and versions (see Table 8, Appendix B.1). Apart from the standard paths, some OEMs place system components on alternative paths like the `vendor/` or `produ ct/` paths. Therefore, we manually verified the location of the targeted TLS stack components (JSSE, JCE, JCA, and the trusted CA store) by manually inspecting the AOSP source code and various OEM images, as well as proprietary binaries or custom implementations (e.g., alternative crypto providers). In cases where discrepancies existed between AOSP and OEM image paths, we recorded the deviations and cross-verified these paths with the corresponding metadata, as exemplified in Table 8. This process ensured that the identified components were accurately mapped and aligned with the expected structures based on AOSP documentation and OEM-specific adjustments.

TABLE 2: Dataset summary across Android versions from 2018 to 2024 with market share (MS) data sourced from StatCounter Global Stats (October 2024) [90].

Version	#Devices	#Models	#Vendors	Release	MS
9	18,178	4,897	221	08/2018	4.25%
10	16,867	5,282	206	09/2019	6.92%
11	12,078	4,068	195	09/2020	12.71%
12	3,942	1,677	121	10/2021	14.27%
13	1,609	689	82	08/2022	19.87%
14	509	171	37	10/2023	33.67%
15	6	5	3	09/2024	-
Total	53,189	16,789	370	-	-

The `build.prop` file, which contains product-specific properties and is typically located in the `/sys tem` path, was available for 16% of the devices in our dataset. We use those devices as ground truth to validate our baseline establishment process in §4.2. We extracted the JSSE and JCE components using their package names, but for JCA components, we took additional steps to account for similarities between BoringSSL and OpenSSL (§2.2). To find alternative JCA providers and ensure fair comparisons between OEM BoringSSL distributions and AOSP, we examined the source files compiled into the shared objects and their strings (§5.2).

Dataset description and Android version coverage. We successfully extracted JSSE, JCA, and JCE components and trusted root CA stores from 53,189 Android devices, covering 16,789 models from 370 OEMs running Android 9 to 15. Our focus on Android 9 and above allows us to track the evolution of OEM customizations over time, while also covering 91.69% of today's Android market (see Table 2). This includes market leaders like Samsung and lesser-known OEMs like Razer. As of Q3-2024, the number of Android 15 devices remains limited.

Ethical considerations. Both Android Dumps and FirmwareScanner datasets are contributed by volunteers who provide their images willingly and with informed consent. FirmwareScanner also collects device metadata (e.g., model, manufacturer, supported SDK version, and `BUILD_FINGERPRINT`) [3].[1] To ensure privacy and detect duplicates, FirmwareScanner uses the MD5 hash of the device's IMEI. This method has been approved by our IRB, ensuring that user privacy is protected. Our study is classified as non-human subjects research.

[1]The `BUILD_FINGERPRINT` is a standard device version identifier structured as: `$(BRAND)/$(PRODUCT)/$(DEVICE):$(RELEASE )/$(BUILD_ID)/$(INCREMENTAL):$(TYPE)/$(TAGS)`.

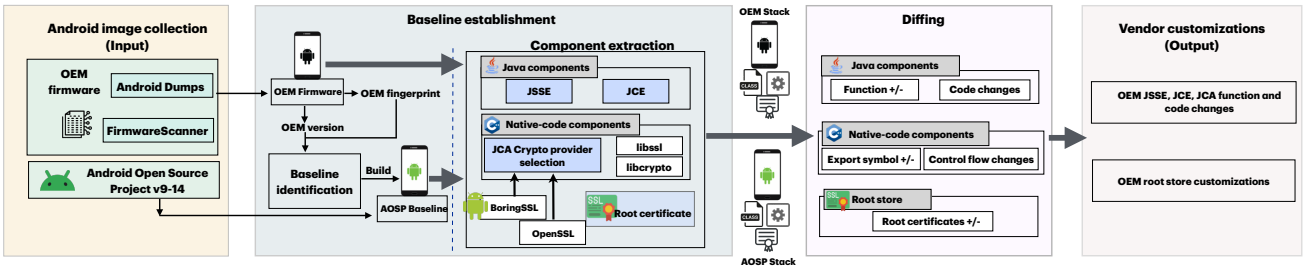

Figure 2: Overview of the complete methodology

4.2. Baseline Identification

Identifying OEM customizations using code diffing techniques requires determining the official AOSP release corresponding to each device model and version [24]. To do this, we rely on two key metadata values collected by FirmwareScanner and Android Dumps: the device's Android version and the `BUILD_FINGERPRINT` (referred to as FG). If the OEM adheres to AOSP build guidelines [39], the `BUILD_ID` within the FG should indicate the baseline version. For instance, a Samsung device with an FG `samsung/j6ltedtvvj/j6lte:10/qp1a.190711.020/j600gtvjuacuh1/release-keys` reports `qp1a.190711.020` as the `BUILD_ID`, which maps to `android-10.0.0_r2`, as documented by Google [39].

However, baseline identification is not always straightforward due to frequent non-compliance with CDD guidelines by OEMs (§7): 58% of devices have `BUILD_ID` tags that do not correspond to known Android releases. In such cases, we prioritize the Android version claimed by the OEM collected by the FirmwareScanner using the `android.os.Build` API, although this can be misleading. To resolve inconsistencies, we employ the following approaches:

1) MIUI-based[2] devices often define custom build IDs like `RKQ1.*.002`, which do not map to any specific Android release. For example, this is seen in Xiaomi 11 using Qualcomm's Snapdragon 865 processor. Similar cases exist in Nokia, OnePlus, and LG models. In such cases, as proposed in prior work [59, 84], we rely on the compilation date from the OEM software repository for the `BUILD_ID`, when available [41], or use the OS version claimed by the OEM.
2) Some models report a build ID that corresponds to a different Android version than the one running on the handset. Since it is unclear if this is intentional, we consider the version claimed by the OEM.
3) Security patches can introduce mismatches. While patches may trigger new AOSP releases, vendors often selectively or partially incorporate them into their devices to ensure legacy feature compatibility or release them with a delay [35, 66]. For instance, Blackview devices built on 2022-04-12 (supporting SDK version 29 for Android 10) already include features from later Android versions. However, their build ID remains as `android-10.0.0_r2`, released on 2019-09-05. In these cases, we use the build ID to set the baseline, simplifying patch detection in our diffing process.

To validate our baseline selection, we leveraged `build.prop` files from 8,573 devices in our dataset. Only 12 devices (0.14%) exhibited conflicting SDK and OS versions, suggesting potential misreporting by OEMs. For all other devices, our method accurately aligned the extracted metadata with known AOSP releases, reinforcing confidence in our baseline selection process.

Beyond this broad validation, during our analysis, we specifically focused on 73 flagged devices where version mismatches suggested partial backporting of security updates. For example, during diffing analysis, some devices exhibited discrepancies where JSSE was up-to-date, but Conscrypt remained outdated, indicating that parts of the TLS stack were not updated in sync with the OS version. Such inconsistencies and OEM practices could lead to incorrect baseline selections and difficulties in result contextualization. However, our combination of automated cross-referencing and manual inspection minimizes incorrect baseline assignments and ensures that our analysis focuses on findings with high confidence.

4.3. Differential Code Analysis

We conduct differential code analysis (diffing) on collected device images to identify OEM customizations relative to their corresponding AOSP baseline.[3] We decompile Java-based components (JSSE and JCE) and binary objects (JCA) into manageable representations as discussed in §4.3.1 and §4.3.2.

Inferring customization purposes. We defined a taxonomy (see Table 3) to categorize OEM customizations according to their impact on TLS protocol properties and security. Yet, the underlying reasons behind such customizations may not always be clear. To address this, we manually inspect AOSP and OpenJDK source code and developer comments to infer functionality, and examine Android security bulletins to check if CVEs are referenced in commits, assessing OEM diligence in applying patches.

4.3.1. Java diffing. We decompile Java-based JSSE and JCE packages to Smali code using baksmali [25] and oat2dex [26]. Smali code is a disassembled representation of the Dalvik executable (dex) bytecode that provides

[2]MIUI is a fully compatible Android-based OS built by Xiaomi for its Xiaomi, Redmi, Poco, and Blackshark devices. MIUI was recently re-branded as HyperOS.

[3]While dynamic analysis would be impractical to achieve this paper's objectives due to its inability to detect and extract customizations at the code level, we use this technique to demonstrate the impact of our findings on user-space apps through Proof-of-Concepts (PoC).

TABLE 3: High-level taxonomy of OEM customizations according to their role on TLS security

Acronym	Category	Description
CONN	Connection and protocol management	Functions for secure connection establishment, including selection and implementation of TLS parameters to mitigate risks of downgrade attacks.
ACP	Alternative cryptographic providers	Use of alternative cryptographic providers like OpenSSL and BouncyCastle.
CFC	Cryptographic functions and configurations	Implementation and configuration of cryptographic algorithms, key management, and cipher suites.
CryptP	Cryptographic provider management	Management and validation of cryptographic providers to ensure the use of secure and trusted providers.
TRS	Trusted root store	Management of trusted root certificates for verifying authenticity of endpoints.
CertM	Certificate management	Developer support for managing certificates, including validation, blocklisting, pinning, and revocation checking to ensure secure and trusted communications.
EV	Endpoint verification	Mechanisms for verifying endpoint identity and integrity, including hostname and certificate validation.

insights into the actual instructions executed during runtime. For each Java component, we apply our code diffing algorithms to automatically identify changes both in class signatures (e.g., function additions or removals) and function implementations; Our pipeline classifies the code changes into categories focusing on various programming functions, including branching control flow (e.g., "if" and "goto"); arithmetic and data manipulation (e.g., "add" and "mul"); method invocation (i.e., "invoke" commands); exception handling (e.g., "try-catch" structures); object manipulation (e.g., "new-instance"); and synchronization (i.e., "monitor" operations). The scale of the analysis prevents us from manually inspecting every customized function to detect logic-level customizations. Therefore, we prioritize functions that, according to OpenJDK's documentation, are most likely to influence TLS security.

Handling obfuscation and compiler artifacts. Identifying TLS customizations through smali diffing presents two challenges: obfuscation and compiler-introduced functions. We observed obfuscation in 12% of analyzed samples, where OEMs demonstrate techniques such as junk-code insertion, class splitting, function renaming, and synthetic accessor functions. These techniques seen in Huawei, Samsung, Xiaomi, and Nokia firmware, alter code structure without modifying functionality.

Junk-code insertion inflates function sizes with redundant operations (e.g., "nop" instructions, excess conditional branches), making diffing more complex. Class splitting distributes functions across multiple classes, further complicating function matching and customization tracking. Additionally, synthetic accessor functions prefixed with "access$$", "-get", or "-wrap" add noise to diffing, making it harder to distinguish OEM customizations from all identified customizations.

To minimize obfuscation impact, we apply keyword-based detection to filter out non-relevant synthetic accessors. For function renaming, we compare logic implementations to detect signature changes without functional differences. Our diffing pipeline automatically merges split classes, ensuring proper reconstruction before analysis. In the cases analyzed in this paper, we manually verify that obfuscation does not misrepresent OEM customizations.

4.3.2. Binary object diffing. We decompile JCA native libraries such as BoringSSL for analysis, specifically: (a) export symbol diffing, and (b) control-flow diffing. Export symbol diffing reveals functions that are added or deleted, while control-flow diffing detects the number of non-matching basic blocks in Control Flow Graphs (CFGs) compared to AOSP. If this number is significant (i.e., at least five blocks), we manually inspect the changes.

In total, we extracted and analyzed 1,034 unique (908,338 in total) binary objects. We use IDA Pro [27] along with the Diaphora [67] plugin in headless mode for decompiling the shared object files into C-like code and performing the diffing analysis, respectively. The results are filtered out using heuristics to eliminate code or function likely introduced by the decompiler or the compiler. We exclude functions (i) without a symbol name in the binary, (ii) that start with an underscore, (iii) related to the `unwind` library, or (iv) that belong to either ARM's Application Binary Interface [40] or the GNU toolchain [48]. We manually integrate into our filter a list of (likely) hardware-specific functions used by high-level languages that, for example, provide better alternatives for arithmetic operations. These heuristics have a very small impact on the number of filtered-out functions, varying from 2% to 6% per object.

4.3.3. Trusted root CA store diffing. We use the OpenSSL toolkit [80] to extract metadata from the 68,385 X.509 certificates found in the trusted root CA stores of devices in our dataset. By comparing these extracted X.509 certificates with those found in AOSP versions 9 to 14, we find certificates removed by Google at each release due to expiration, compatibility issues, or loss of trust, as well as those added by OEMs in their devices. We note that we only analyze root stores from non-rooted devices located in the default path for root CAs. This allows us to gain a comprehensive view of the root store and mobile PKI landscape across OEMs, spanning multiple Android versions, and detect certificates appearing in a specific set of OEMs. We investigate whether unremoved or newly added CAs were excluded from AOSP due to a lack of trust by correlating CA information with news articles, AOSP change logs, and Mozilla's TLS Observatory, all of which document security events leading to the distrust of these CAs. Furthermore, we analyze OEMs shipping already expired certificates by the OS version release date.

5. Results

Our code diffing analysis shows that 80% of OEMs modify at least one model from its corresponding AOSP codebase. In this section, we address RQ1 by analyzing these deviations across different TLS components, assessing their security risks using the taxonomy in Table 3, covering protocol configuration to endpoint verification.

TABLE 4: Unique OEM models with customizations impacting each TLS security function category defined in Table 3. In parentheses, we report the number of OEMs.

Category	JSSE	JCE		JCA		Root Store
		OkHttp	Conscr.	libssl	libcryp.	
CONN	67 (22)	1,940 (42)	755 (11)	-	-	-
CFC	-	-	-	67 (9)	178 (36)	-
CryptP	16 (6)	6 (2)	760 (11)	-	-	-
TRS	-	-	-	-	-	2,068 (274)
CertM	1,262 (7)	6 (2)	71 (12)	48 (8)	74 (12)	-
EV	16 (6)	1,707 (62)	15 (8)	-	-	-

```
% AndroidPlatform class
public boolean isCleartextTrafficPermitted(String hostname) {
    try {
      Class<?> networkPolicyClass = Class.forName("android.security.
            NetworkSecurityPolicy");
      Method getInstanceMethod = networkPolicyClass.getMethod("getInstance");
      Object networkSecurityPolicy = getInstanceMethod.invoke(null);
      Method isCleartextTrafficPermittedMethod = networkPolicyClass
          .getMethod("isCleartextTrafficPermitted", String.class);
      return (boolean) isCleartextTrafficPermittedMethod.invoke(
            networkSecurityPolicy, hostname);
    } catch (ClassNotFoundException | NoSuchMethodException e) {
     return super.isCleartextTrafficPermitted(hostname);
    } catch (IllegalAccessException | IllegalArgumentException |
          InvocationTargetException e) {
      throw new AssertionError();
    }
}

% Platform class
public boolean isCleartextTrafficPermitted(String hostname) { return true; }
```

Figure 3: Exception handling in `AndroidPlatform` class executes `isCleartextTrafficPermitted`.

Overall, we identify 1,144 customized TLS APIs and functions that are either absent, added, or logically modified in OEM TLS stacks. To systematically evaluate their impact, we map the findings to threat scenarios defined in Table 1, TN1 (Outdated TLS components), TN2 (Weak encryption), TN3 (Insecure endpoint validation), and TA1 (App-level exploits). This framework allows us to assess how specific TLS modifications increase exposure to network adversaries and malicious apps. Section 6 evaluates how the customized APIs compromise network security guarantees of mobile apps on the Google Play Store. Our results suggest that, while Google-certified OEMs generally preserve JSSE class signatures for compatibility (§7), all OEMs often customize lower-layer components within JCE and JCA, as Table 4 shows. In contrast, non-certified OEMs modify all layers of the stack.

5.1. Protocol and Connection Management

Our analysis reveals that 15.4% of OEMs modify critical functions for TLS connection establishment and protocol management in OkHttp (13.2% of OEMs) and Conscrypt (3.5% of OEMs), as we describe next:

Ignoring clear-text traffic blocking policies: Android allows developers to enforce app-specific policies using the `isClearTextTrafficPermitted` boolean tag in the manifest. When set to `false`, the OS blocks any plaintext HTTP connections using the `isClearTextTrafficPermitted()` method.

Our analysis reveals that MIUI Android 9 to 14 models include a distinctive OkHttp distribution —*"miui-okhttp"*, based on Square's [89]—that differs from AOSP in class structure and function implementations. Notably, its `isCleartextTrafficPermitted` method always returns `true` in the likely triggerable exception (see Figure 3), disregarding developer-defined encryption policies in MIUI devices (TN2, TA1). This poses a serious risk, particularly for third-party advertising SDKs, where developers have no control over data transmission [54, 87]. We developed a Proof-of-Concept to demonstrate this vulnerability and responsibly disclosed it to Google's Android team [63], which escalated the issue to Xiaomi [97]. The case is under investigation, and we have agreed to keep it confidential until April 17th, 2025.

Deprecated protocol support: We observe OEMs inconsistently supporting deprecated AOSP OkHttp functions for TLS protocol configurations. For example, the Huawei Hol-U19 model running Android 9 keeps the `supportTlsIntolerantServer` function to establish connections with endpoints supporting the deprecated and insecure SSLv3 standard, which, if invoked, may expose apps to the POODLE attack (TN1) [46, 68]. On a positive note, later Huawei models follow AOSP.

Setting TLS protocol preferences: Five OEMs, including Samsung and Redmi, customize JSSE, OkHttp, and Conscrypt functions for setting TLS protocol preferences. For example, Samsung's Galaxy Note 7 and 10+, and Yestel's X7-EEA, all running Android 9, omit classes added in Android 7 like `ConnectionSpec` (OkHttp) with `allEnabledTlsVersions` and `allEnabledCipherSuites` functions, that enable app developers to use the most recent TLS protocol versions and cipher suites. As a result, apps on these devices may remain exposed to outdated encryption risks (TN1, TN2) [5].

ALPN extension support: Conscrypt's `selectApplicationProtocol` method in the `ConscryptEngine` class allows developers to manage performance-security trade-offs by enabling protocols like HTTP/2 and SPDY through the TLS extension Application-Layer Protocol Negotiation (ALPN). The ability to explicitly define ALPN settings prevents attacks like the ALPACA attack [2], i.e., forwarding a connection to a different server (e.g., HTTP to FTP). Although developers can still use HTTP/2 without ALPN, they lose the advantage of automatic protocol negotiation, leading to sub-optimal security configurations. This function, integrated into AOSP version 11, is missing in Android 11 versions of Redmi Sweet-EEA and the Samsung A50 and A70 models.

5.2. Alternative Cryptographic Providers

To ensure the safe usage of cryptographic providers, OEMs must properly maintain Conscrypt and BoringSSL components [36]. However, many OEMs incorporate additional cryptographic providers in their products, either for legacy support or to enhance security:

- Oppo, Realme, and Alps use OpenSSL alongside BoringSSL to support legacy features and a broader feature set through the `CompatibilityHelper` (Listing 9). Unfortunately, these devices use outdated versions—`1.0.1j` (October 15, 2014), and `1.0.1e` (February

11, 2013)—which are vulnerable to over 60 known vulnerabilities like CVE-2016-6304, exposing devices to potential DoS attacks (TN1) [44].
- LGE and HTC (Android 9-13) integrate Libgcrypt, a cryptographic library implementing robust functions like SHA-256 and SHA-512 [28], used in products like Google Home (Listing 10). Though the number of CVEs is not a direct measure of security, Libgcrypt has had a lower number of assigned CVEs compared to OpenSSL, with 15 since 2013 [45].
- Samsung adds its proprietary Knox cryptographic libraries, i.e., `libknox_encryption` and `libsecpkcs11_engine.secsmartcard.samsung`, which work with TrustZone to secure sensitive data and support smartcard-based encryption [16] (Listing 11).
- Finally, 18 OEMs integrate `libsoftkeymaster`, a software-based fallback for devices without hardware-backed keystores, ensuring foundational security for cryptographic operations and key management.
- Xiaomi, Redmi, and Poco extend BoringSSL with `libadvanced_crypto` for key management and digital signatures for secure operations like device tracking in Xiaomi's Find Device app (Listing 12).

5.3. Cryptographic Functions and Configurations

Over 180 device models (36 OEMs) customize their BoringSSL distribution, leading to deviations from AOSP's cryptographic configurations. While some changes may be performance-driven or intended for compatibility, others introduce security inconsistencies.

Cipher suite support: Six OEMs, including Vivo and Blu, omit critical functions like `EVP_aead_aes_256_gcm_tls13`, which define TLSv1.3 cipher suites in at least 14 Android 10 and 11 models. This impacts TLS security, as AES-GCM is widely used [68]. We note that this removal may be deliberate, as devices lacking AES hardware acceleration may benefit performance-wise from using ChaCha20 [18]. Other Android 9 and 10 models from Alps, Blu, Huawei, and Samsung offer deprecated TLS functions like `SSL_CIPHER_is_AESGCM` and `SSL_CIPHER_is_CHACHA20POLY1305`, all removed in Android 9. While their presence does not inherently introduce vulnerabilities, using outdated functions in modern implementations could weaken security (TN1).

TLS cryptographic configurations: Several devices, such as the Xiaomi POCO X3 Pro (Android 12), all OnePlus models (Android 12-14), and Redmi Sweet EEA (Android 13), lack two critical functions:

- The `SSL_set_enable_ech_grease` function implements the Generate Random Extensions And Sustain Extensibility (GREASE) mechanism in the Encrypted Client Hello (ECH) extension. While ECH GREASE does not directly protect SNI values, disabling it could increase the risk of apps being affected by SNI-based filtering. Some network intermediaries may misinterpret GREASE extensions as anomalies, leading to unnecessary blocking [88]. Therefore, some OEMs might disable this feature in response to such blocking and limited consideration of its implications.
- The `SSL_add_application_settings` function manages ALPN, enabling secure protocol handling like HTTP/2. While ALPN is primarily a performance optimization rather than a security control, its removal limits developer flexibility in handling secure connections [2].

```
% Standard AOSP implementation
invoke-static {}, Lsun/security/jca/Providers;->getFullProviderList()Lsun/
       security/jca/ProviderList;
move-result-object v2
add-int/lit8 v4, p1, -0x1
invoke-static {v2, p0, v4}, L
sun/security/jca/ProviderList;->insertAt(Lsun/security/jca/ProviderList;Ljava/
       security/Provider;I)Lsun/security/jca/ProviderList;

% Modified implementation with FIPS check for Zebra TC21 model
invoke-virtual {v1}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
move-result v2
if-eqz v2, :cond_1d
invoke-static {}, Ljava/security/Security;->isDeviceSupportFIPS()Z
move-result v2
if-nez v2, :cond_1d
const-string v2, "Device will Not Support SCFIPS"
invoke-static {v2}, Ljava/lang/System;->logE(Ljava/lang/String;)V
:try_end_1b
.catchall {:try_start_3 .. :try_end_1b} :catchall_39
monitor-exit v0
return v3
```

Figure 4: Comparison of provider integration methods. On top, the standard AOSP method for integrating providers. Below, the Zebra TC21 modification for FIPS support.

Other devices similarly lack key functions for cryptographic configuration and management:

- Several Realme and Samsung Galaxy Android 9-10 models lack cryptographic functions, `EVP_aead_aes_128_gcm_siv` and `EVP_aead_aes_256_gcm_siv` introduced in AOSP following RFC 8452 to mitigate AES encryption attacks in GCM-SIV mode [72]. While AES-GCM-SIV offers nonce misuse resistance, its absence may indicate performance trade-offs.
- Several Oppo and OnePlus models running Android 13 lack the `EVP_hpke_x25519_hkdf_sha256` function implementing HPKE using X25519 and HKDF-SHA256. This function combines X25519 for fast key generation with HKDF-SHA256 to produce strong encryption keys, which is important for modern key exchange protocols, including TLS 1.3 ECH.

5.4. Cryptographic Provider Management

Cryptographic provider management allows developers to select, configure, and maintain secure cryptographic providers for TLS connections. Our analysis finds at least 14 OEMs with either security downgrades or enhancements in the Java layers and 13 OEMs introducing errors in their Conscrypt JNI implementations.

Legacy cryptographic provider support: Two Samsung models implement the deprecated functions `getInstanceFromCryptoProvider` and `setSdkTargetForCryptoProviderWorkaround`. These functions manage access to the deprecated `CryptoProvider` and were removed from AOSP in version 7 as it is vulnerable to the "SecureRandom" bug (TN1) [82].

Ciphersuite order preference: Conscrypt's `SSLParametersImpl` class introduces the `setUseCipherSuitesOrder` function in AOSP version 7 to allow developers to override the default cipher suites and protocol parameters defined in JSSE (TN2). However, this function is missing in the Allwinner T3 running Android 10.

FIPS support: The Zebra TC21 (Android 10) includes the `isDeviceSupportFIPS` function to verify whether

```
% Vulnerable JNI implementation
EVP_CIPHER * FUN_0000a2dc(undefined4 param_1)
{
  char *local_14;
  FUN_0000a14e(auStack_1c,param_1);
  if (strcasecmp(local_14,"rc4") == 0) return EVP_rc4();
  if (strcasecmp(local_14,"des-cbc") == 0 || strcasecmp(local_14,"des-ede-cbc
        ") == 0)
      return EVP_des_cbc();
}

% AOSP patch for CVE-2016-6709
EVP_CIPHER * NativeCrypto_EVP_get_cipherbyname(_JNIEnv *param_1, _jclass *
       param_2, _jstring *param_3)
{
  char *__s1 = (**(code **)(*(int *)param_1 + 0x2a4))(param_1,param_3,0);
  if (strcasecmp(__s1,"rc4") == 0) return EVP_rc4();
  if (strcasecmp(__s1,"des-cbc") == 0) return EVP_des_cbc();
  if (strcasecmp(__s1,"des-ede-cbc") == 0) return EVP_des_ede_cbc();
}
```

Figure 5: Conscrypt JNI vulnerability, CVE-2016-6709. On top, the incorrect mapping of 2-key 3DES to regular DES leads to information disclosure. Below, the patch implemented in AOSP.

the BouncyCastle complies with the NIST FIPS standard (see Figure 4). While AOSP only allows provider insertion, Zebra introduces this method to meet Common Criteria certifications and security requirements from government agencies [99]. This device also includes StripyCastle [43], a FIPS-compliant provider, further supporting the inclusion of `isDeviceSupportFIPS`.

Conscrypt JNI implementation: Conscrypt's JNI code analysis reveals that the `libjavacrypto.so` present in 30 OEM devices, remains vulnerable to CVE-2016-6709 [11]. This flow could expose sensitive data when an app uses legacy encryption APIs (TN1). Although this vulnerability was fixed in Android versions 6.0, 6.0.1, and 7.0, certain Alps models and even Google-certified models from Samsung and BMXC still use outdated JNIs, alongside OpenSSL's X.509 and BoringSSL (see Figure 5).

5.5. Trusted Root Store

The trusted root store contains default trusted certificates for a device. We found 274 OEMs who modified the root store by adding or removing certificates.

Non-AOSP certificates: Several OEMs add root CA certificates issued by organizations like UserTrust Network, Digital Signature Trust, and SecureSign, which are untrusted by AOSP and audited root stores like Mozilla. For instance, the Alps Android 9 root store contains 84 non-AOSP certificates, 18 of which were expired upon release. The additions include certificates issued by TrustCor, a CA with ties to US intelligence services, distrusted by Mozilla and Microsoft for signing malicious certificates [93]. On average, OEMs add 25 non-AOSP certificates to Android 12, while non-certified OEMs add more.

Presence of removed and expired certificates: We found 117 OEMs including 69 AOSP-removed certificates in their trusted root CA store. "AOSP-removed certificates" are those present in AOSP version X, but removed in subsequent AOSP versions X+1 due to expiration or trust issues. In 85% of OEMs, these certificates persist in later Android versions (X+1, X+2, etc.), showing failure in root store updates (see Figure 6). We also find cases where AOSP removed certificates after version 8, yet these remain in use on devices running versions 12-14. Prior

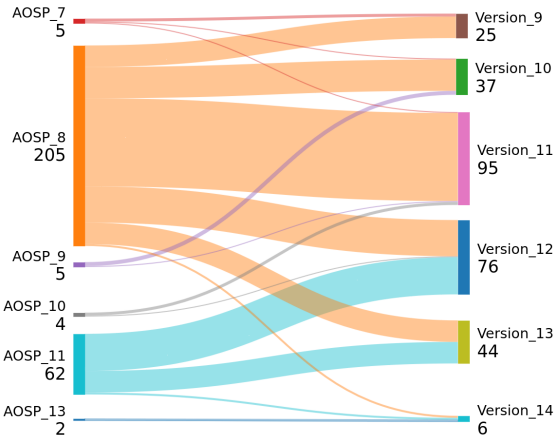

Figure 6: Number of OEMs retaining certificates removed from AOSP in previous versions. The left side shows the last AOSP version in which a certificate was included, while the right side represents later OEM versions where the same certificate is still present. This indicates the number of OEMs that did not update their root stores.

work links expired certificates in websites to poor security posture and outdated servers with known CVEs [83].

Untrusted certificates: Beyond the TrustCor case mentioned above, we find well-known untrusted root certificates in several Android models. Two Comodo certificates, removed after Android 8, remain in 116 OEMs, including major brands like Samsung, Sony, LGE, and HTC. These certificates were compromised in a 2011 incident when an Iranian threat actor breached Comodo's infrastructure and issued malicious certificates for Google, Yahoo, and Microsoft [96]. Consequently, Comodo lost trust within the PKI ecosystem, and its certificates were removed from the trusted root stores of major browsers. Yet, we find these certificates across all Android versions in our dataset, with 44% found on Android 11 devices (2020). Their continued presence is concerning (TN3), as connections using such certificates will be treated as "normal."

5.6. Certificate Management

The widespread presence of rogue, expired, and untrusted certificates in root CA stores requires app developers to verify TLS certificate validity proactively. However, 12 OEMs remove key JSSE, Conscrypt, BoringSSL, and OkHttp functions for managing certificates, impacting certificate blocklists, pinning verification, chain strength analysis(i.e., key length and signature algorithm security), OCSP (Online Certificate Status Protocol) support, SCT (Signed Certificate Timestamp) verification, and trust chain construction. While these removals are prevalent in Android 9 models, they are also present in some Android 11 devices from Motorola, Infinix, and Poco.

OCSP support. Several devices integrate BoringSSL functions to query OCSP [74] and verify X.509 certificate revocation status. `OCSP_BASICRESP_delete_ext`, `OCSP_response_status`, `d2i_OCSP_P_CRLID`, `OCSP_REVOKEDINFO_new`, and `OCSP_REQ_CTX_free` functions were added into 44 Android 9 and 10 models from Samsung, Advan, Verizon, and Alps. All these func-

tions are taken from OpenSSL, indicating intentional use for compatibility with legacy systems. Yet, apps running on non-certified OEMs like Alps cannot process OCSP responses due to the lack of key functions like `SSL_enable_ocsp_stapling`, making them vulnerable to expired or revoked certificates (TN3) [56]. Similarly, some Samsung models running Android 9-10 also miss the `SSL_set_ocsp_response` function introduced in BoringSSL with Android 8. In these cases, app developers are forced to manually handle certificate status checks, which increases the risk of improper validation (TN3). As a result, revoked certificates might still be accepted during the caching window.

Certificate blocklisting and pinning. Several OEMs lack support or introduce new functions for certificate blocklisting and pinning operations:

- Samsung Galaxy S23 and Yestel X7-EEA miss Conscrypt's `checkChainPinning` function for checking if a certificate chain includes pinned certificates.
- Allwinner T3 model running Android 10 lacks critical Conscrypt functions, including `isCTVerificationRequired`, `checkKeyLength`, `checkSignatureAlgorithm`, `getOCSPResponse`, `setOcspResponses`, `isCTVerificationEnabled`, and `verifyChain`. Without these functions, app developers face limitations in secure certificate management, reducing their ability to enforce proper certificate validation. As a result, apps may unintentionally accept fraudulent or improperly validated certificates, making them vulnerable to person-in-the-middle attacks (TN3) [47, 56].
- Some Alps models running Android 10 lack the OkHttp `setCertificatePinner` and `getCertificatePinner` methods to specify trusted CAs. Therefore, apps must rely on the default root CAs chosen by the OEM (TN3) [53]. This is concerning due to the presence of untrusted certificates in the Alps' root stores (§5.5).
- Honeywell has taken proactive steps to enhance the device and user safety. For instance, Honeywell Android 12 models use the `getForceDisabledSystemCert` function within Conscrypt's `TrustedCertificateStore` class, to retrieve system certificates explicitly marked as "force disabled." These certificates are blocked from being trusted or reinstalled, thereby protecting the system from trusting disabled certificates.

5.7. Endpoint Verification

Endpoint verification is crucial for ensuring that TLS clients connect only to trusted and verified endpoints. We find 1,721 device models with varying implementations of endpoint verification across JSSE and JCE. We next describe some relevant cases:

- The Android 10 Allwinner T3 model lacks the `setDefaultHostnameVerifier` method that prevents developers from overriding the default `HttpsURLConnection` hostname verifier. This limitation makes it challenging to enforce custom hostname verification rules stricter than those outlined in RFC 2818 (TN3) [8].
- Several Samsung and Yestel models running Android 9 lack the OkHttp `setDNS` method, limiting app developers' ability to override the default DNS configuration, hence increasing the risk of DNS spoofing (TN3) [64].
- In its Android 12 release, OkHttp introduced the `isPrintableAscii` method in the `OkHostnameVerifier` class to address CVE-2021-0341 [12]. This function rejects non-ASCII hostnames and Subject Alternative Names (SANs) to ensure that domain names in X.509 certificates follow IDNA 2008 rules, thus protecting apps from DNS spoofing and name collision exploits. Interestingly, several Android 9 to 11 models from Samsung, Vivo, Nokia, and Xiaomi already incorporate this function, indicating that some OEMs take proactive patching measures against this attack.

5.8. Other Interesting Customizations

Vivo devices running Android 13 introduce two security-enhancing features that bridge TLS APIs with the Android permission model to block unauthorized access to permission-protected system resources:

- The `isEmail` function in JSSE's `Socket` class verifies whether a port number matches standard email protocols—SMTP (25), POP3 (110), or IMAP (143). Consequently, only if the app has been granted the dangerous `SEND_EMAIL` permission to access email services will the connection be authorized.
- The `NetworkInterface.shouldProtectMac` function adds a conditional control layer for accessing the MAC address of the `wlan0` (WiFi) interface. While Android 10 already restricts MAC address access to system apps, Vivo's implementation integrates `VivoPermissionManager` for extra security checks. Specifically, Vivo ensures that the returned MAC address of any interface is protected through randomization if the location of the users is China, WiFi randomization address is not set, or the app requesting access is a system app using the `isOverseas` method (see Figure 8). Interestingly, the method is implemented using reflection, which opens additional security concerns [57] as the policy could be adjusted or bypassed dynamically. We are unable to explain the reason behind this discrimination against Chinese customers.

6. Impact of Customizations on Developers

The OEM customizations reported in §5 introduce TLS API fragmentation and inconsistencies that could cause user-installed apps to raise exceptions, behave unpredictably, or crash. In this section, we explore RQ2 by analyzing how these deviations impact real-world app functionality and security. The most critical modified TLS methods identified in §5 influence the following areas:

1) Enforcing encrypted traffic (e.g., OkHttp's `isHttps`), which, if not properly handled, leaves apps vulnerable to eavesdropping or person-in-the-middle attacks across approximately 1,900 Android models.
2) Defining TLS parameters (e.g., JSSE's `setSSLParameters`), making it hard for developers to configure secure protocol negotiations, potentially leaving users of around 760 models exposed to protocol downgrade and cross-protocol attacks like POODLE.
3) Performing certificate (e.g., JSSE's `X509Certificate.verify`) and endpoint (e.g., JSSE's `setSNIMatchers`) validation, weakening apps' ability to verify

TABLE 5: Usage of missing TLS APIs by Android apps.

Impact	API	#Apps (Installs)
JSSE		
CertM	L/java/security/cert/X509Certificate/verify	14,520 (676B)
CFC	Ljava/security/spec/EllipticCurve/checkValidity	9,705 (340B)
EV	Ljavax/net/ssl/SSLServerSocket/setSSLParameters	735 (10B)
EV	Ljavax/net/ssl/SSLParameters/setEndpointIdentificationAlgorithm	765 (14B)
EV	Ljavax/net/ssl/SSLServerSocket/getSSLParameters	706 (9B)
EV	Ljavax/net/ssl/SSLParameters/setServerNames	338 (7B)
CONN	Ljavax/net/ssl/SSLParameters/setAlgorithmConstraints	99 (6B)
CryptP	Ljavax/net/ssl/SSLParameters/setUseCipherSuitesOrder	94 (1B)
EV	Ljavax/net/ssl/SSLParameters/setSNIMatchers	91 (1B)
EV	Ljavax/net/ssl/SSLParameters/getEndpointIdentificationAlgorithm	48 (1B)
CONN	Ljavax/net/ssl/SSLParameters/getAlgorithmConstraints	21 (949M)
EV	Ljavax/net/ssl/SSLParameters/getServerNames	24 (144M)
CertM	Ljava/security/cert/CertPathValidator/getRevocationChecker	41 (292M)
CertM	Ljava/security/cert/CertPathBuilder/getRevocationChecker	41 (292M)
EV	Ljavax/net/ssl/SSLParameters/getSNIMatchers	17 (345M)
OkHttp		
CONN	Lokhttp/Response/handshake	1,389 (14B)
CONN	Lokhttp/Request/isHttps	1,200 (28B)
CONN	Lokhttp/OkHttpClient/getDefaultSSLSocketFactory	1,184 (4B)
CONN	Lokhttp/internal/Platform/configureTlsExtensions	1,151 (4B)
CertM	Lokhttp/OkHttpClient/getCertificatePinner	1,092 (23B)
CertM	Lokhttp/Address/getCertificatePinner	1,092 (23B)
CONN	Lokhttp/OkHttpClient/getProtocols	1,092 (23B)
CONN	Lokhttp/Address/getProtocols	1,092 (23B)
CertM	Lokhttp/internal/Platform/trustRootIndex	934 (5B)
CONN	Lokhttp/OkHttpClient/setFollowRedirects	136 (334M)
CONN	Lokhttp/internal/platform/Platform/isClearTextTrafficPermitted	112 (898M)
EV	Lokhttp/internal/tls/OkHostnameVerifier/allSubjectAltNames	61 (394M)
CONN	Lokhttp/internal/Platform/getSelectedProtocol	61 (394M)
CONN	Lokhttp/Route/requiresTunnel	61 (394M)
CONN	Lokhttp/OkHttpClient/setConnectionSpecs	84 (154M)
CONN	Lokhttp/Connection/getHandshake	14 (120M)
CONN	Lokhttp/OkHttpClient/setProtocols	72 (86M)
EV	Lokhttp/OkHttpClient/setDns	6 (53M)
CryptP	Lokhttp/ConnectionSpec/allEnabledCipherSuites	1 (904K)
CONN	Lokhttp/ConnectionSpec/allEnabledTlsVersions	1 (904K)
Conscrypt		
CONN	Lcom/android/org/conscrypt/Conscrypt/exportKeyingMaterial	1 (666K)

certificates and endpoints in about 1,200 devices. This weakness is amplified by the presence of untrusted root certificates (e.g., TrustCor and Comodo) in 274 OEMs (74% of all OEMs), increasing network security risks.

To estimate the number of affected apps, we statically study the code of 20k mobile apps published on Google Play for statements invoking any of the 1,144 customized APIs found in §5. Over 75% of the analyzed apps rely on at least one modified or missing TLS function. Table 5 reports the most used customized APIs, and their impact area (per Table 3), highlighting the number of apps that depend on them and apps' cumulative install count. We observe that APIs for certificate management `X509Certificate.verify` (JSSE) and endpoint verification `Response.handshake` (OkHttp) are invoked by 73% of the analyzed apps, confirming their critical role in ensuring secure communications in the Android ecosystem.

Unfortunately, TLS API customizations force developers to implement complex workarounds for certificate verification on phones without these methods or lacking recent TLS extensions or standards, which can increase security risks. More importantly, developers unaware of these undocumented modifications simply expose their users and apps to security risks across Android devices.

6.1. Case studies

Below, we present examples of six relevant apps that demonstrate how TLS API fragmentation affects functionality and security. We triage and manually examine the cases with the highest potential security impact according to AOSP documentation, focusing on how developers mitigate issues. This section does not aim to provide a comprehensive function dependency analysis but highlights how developers handle or fail to address API fragmentation.

Potential functional and security errors:

- Board Kings (52M installs). This game uses the `checkValidity` function to validate server-side certificates before proceeding with encryption operations (see Figure 13). If the certificate is invalid, it throws an exception caught and re-thrown as `InvalidKeyException`, preventing encryption with invalid certificates and allowing graceful error handling. Without `checkValidity`, the app risks failure as the app developer has not accounted for this scenario.
- Xiaomi Home (59.5M installs). This app uses the `handshake` function in OkHttp's `Response` class to securely access SSL/TLS handshake parameters, including negotiated protocol and cipher suite (see Figure 14). Without this function, the app risks compromising connection security, as the `readNetworkResponse` method fails to retrieve handshake details. Since this function lacks specific exception handling, the app may unknowingly use insecure encryption.
- Mintegral SDK. This popular advertising SDK— package name `com.mbridge.msdk`— calls `isCleartextTrafficPermitted` function from `Platform` class in OkHttp to check the clear-text policy defined by the app developers on the Android Manifest file. In the case of MIUI devices (§5.1), this function always returns True, allowing clear-text traffic for all hostnames. When `RealConnection.connect()` invokes this function, clear-text traffic will be allowed, thus exposing users' data to eavesdropping (see Figure 15). When `isCleartextTrafficPermitted` returns True, the `connect` method does not throw an exception for clear-text traffic. This finding confirms that the OkHttp vulnerability found on MIUI devices may impact a large number of users, given Mintegral's estimated high presence in mobile apps [87].

Correct fragmentation handling and management:

- Temple Run 2 (1.05B installs). This game exemplifies developers' awareness of TLS fragmentation issues and good practices. It calls the `verify` method (class `java.security.spec.EllipticCurve`) to ensure a certificate is correctly signed by its issuer (see Figure 16) as part of the `isValidLink` method for checking certificate chain integrity. If unsupported, it throws a `GeneralSecurityException`, correctly caught by the app, with `isValidLink` returning false.
- IM ShareChat (503M installs). This app uses `setSSLParameters` and `setEndpointIdentificationAlgorithm` methods to enforce TLS parameters and hostname verification (see Figure 17). If unavailable, it catches `NoClassDefFoundError` and `NoSuchMethodError` exceptions, retrying if `httpsHostnameVerificationEnabled` is true; hence keeping a secure connection by enforcing TLS settings and hostname verification even if standard methods are absent.
- Citibank Mobile (10M installs). The app checks certificate revocation status via OCSP using `getRevocationChecker` from `java.security.cert.CertPathValidator` (see Figure 18). If the method is unsupported, the app throws a `NoSuchAlgorithmException`, which it catches and returns false, signaling the failure to perform the OCSP check and proceeding with further certificate validation processes.

Third-party TLS libraries. Some app developers rely

TABLE 6: Samsung (S) and MIUI (M) models deviating from AOSP v9 to v13. Dashes (-) indicate that there are no customizations related to those listed on Table 3 taxonomy.

Version	JSSE		Conscrypt		OkHttp		JCA		Root Store	
	S	M	S	M	S	M	S	M	S	M
9	0.8%	0.3%	0.6%	-	2.1%	100%	3.8%	0.6%	-	3.8%
10	50.6%	-	50.1%	0.5%	53.0%	100%	1.6%	0.2%	-	0.5%
11	30.2%	0.2%	0.1%	-	30.2%	100%	0.1%	-	0.5%	0.9%
12	12.1%	-	-	-	12.1%	100%	-	0.6%	0.9%	7.5%
13	-	-	-	-	-	100%	0.7%	-	2.2%	58.3%

on alternative open-source TLS libraries like Volley [33], Retrofit [32], or Facebook Fizz [52] rather than using standard APIs. In our study of 20k Android apps from the Google Play Store, we find that 18% of the studied apps integrate Volley, 33.5% Retrofit, and only 0.1% use Facebook Fizz. It is unclear, however, if these libraries can reduce the risks caused by API fragmentation. Through code inspection of their open-source projects, we find that newer versions of both Volley and Retrofit still rely on JSSE and OkHttp, which can be customized by OEMs. While these dependencies may now have a limited impact on overall app security, they still raise concerns. The self-contained nature of Fizz (based on OpenSSL) makes it potentially safer for developers.

7. OEM Motivations, Selective Patching, and Compliance Challenges

This section explores potential motivations behind OEM TLS customizations and will examine how these motivations contribute to fragmented security updates and deviations from AOSP standards using real-world case studies from our study. In addressing RQ3, we evaluate the effectiveness of Google's CTS/CDD enforcement and address its inconsistencies. Finally, we discuss potential mitigation strategies including stronger compliance enforcement, transparency tools, and regulatory measures.

7.1. OEM Motivations for TLS Customizations

OEMs could customize the TLS stack for a multitude of reasons, often with the target of balancing security, performance, and compatibility. While some of these modifications might be necessary for device functionality, others introduce fragmentation and security risks. Unfortunately, we do not have public information as to the exact reason for these customizations, yet our analysis identifies three potential primary motivations behind customizations.

Legacy Support and Compatibility: OEMs continue using older cryptographic libraries despite the availability of more secure alternatives. Our analysis identifies multiple devices that still ship with outdated OpenSSL versions, with few retaining support for SSL v3, which was deprecated in 2015 due to known vulnerabilities (§5.1). By analyzing these deviations across multiple Android versions, we observe this trend more in older Android devices, likely due to extended firmware updates that are misaligned with AOSP upstream changes.

Prioritization and Selective Updates: Our analysis reveals that OEMs selectively update TLS components rather than apply uniform updates across all layers and all their devices, leading to fragmented security postures even within the same Android versions and products.

We observe several devices with incomplete transitions from Android 10 to 11, where JSSE aligns with AOSP but Conscrypt is outdated. For example, the Samsung Galaxy A50 (SM-A505G) and A70 (SM-A7050) both exhibit 33 missing Conscrypt function updates. Similarly, Realme RMX3701 (REE2ADL1) running Android 13 shows missing updates in JSSE, while other TLS components remain up-to-date. Since our dataset includes many other models from these OEMs, we can conclude that OEM update strategies are not uniform across their product lines. Instead, our empirical results reveal that patching decisions may be influenced by internal prioritization, resource constraints or product segmentation strategies. Finally, some OEMs disable cryptographic mechanisms for compatibility reasons. For example, the removal of GREASE support—from Oneplus and Redmi devices—for Encrypted Client Hello (ECH) may result from network interoperability concerns, reflecting how usability is sometimes prioritized over enhanced security (§5.3).

Regional Policies: Prior work has shown that OEMs incorporate regional or government requirements primarily within the trusted root store, where certificates are added to comply with local and government regulations [94]. Our analysis shows that these region-specific modifications may extend beyond the root store to core Java packages within the TLS stack. For example, in China, Vivo triggers an additional security measure that randomizes MAC addresses to enhance user privacy (§5.8).

7.2. OEM Patching Practices

AOSP maintainers regularly update the TLS stack to patch vulnerabilities and support new standards. However, our analysis shows that OEMs—except for Google Pixel devices—do not necessarily follow upstream, possibly for the reasons discussed in §7.1. This selective patching results in incomplete security updates across 2,539 models (16%) from 85 OEMs, fragmenting TLS security even among the same Android version. Despite Google's project Treble [19], some OEMs still introduce security-degrading deviations, raising concerns about the overall effectiveness of compliance mechanisms(§7.3).

Case study: Samsung vs. MIUI. We compare the patching strategies of Samsung and MIUI to exemplify varying approaches at both the vendor and product levels. MIUI and Samsung jointly account for over 50% of the global Android market share [9]. Table 6 shows the percentage of models deviating from AOSP for each layer of the TLS protocol stack, broken down by Android version. A general observation is that Samsung and MIUI follow

TABLE 7: CDD violations across OEMs and models.

Policy	% Models (OEMs)	Example
3.1. MUST ship with each and every non-SDK interface on the same restricted lists as provided via the provisional and denylist flags in prebuilts/runtime/appcompat/hiddenapi-flags.csv path for the appropriate API level branch in the AOSP. (Focusing on interface availability)	2.4 (6.7)	Realme RMX3710
3.1. MUST ship with each and every non-SDK interface on the same restricted lists as provided via the provisional and denylist flags in prebuilts/runtime/appcompat/hiddenapi-flags.csv path for the appropriate API level branch in the AOSP.	4.7 (27.1)	Xiaomi Nitrogen
3.1. MUST provide complete implementations, including all documented behaviors, of any documented API exposed by the Android SDK or any API decorated with the "@SystemApi" marker in the upstream Android source code.	1.2 (5.8)	Redmi C55
3.2.2. To provide consistent, meaningful values across device implementations:VERSION.RELEASE, The version of the currently executing Android system, in human-readable format.	<9 (6.0)	Oppo CPH2145
9.8.4. MUST preinstall the same root certificates for the system-trusted Certificate Authority (CA) store as provided in the upstream Android Open Source Project.	36 (85.0)	Samsung A52Q

different strategies: Samsung models show greater variability across all components, whereas MIUI exhibits a consistent approach in customizing OkHttp across their models. Additionally, MIUI shows an increasing number of modifications to the root store across Android versions. Despite the uniform base of MIUI's OkHttp, individual MIUI brands still exhibit distinct customizations, contributing to fragmentation. These differences highlight the lack of a standardized patching approach across OEMs, as Samsung's approach leads to inconsistent patching, while MIUI's centralized structure lacks cross-brand uniformity.

7.3. CDD and CTS Effectiveness

Certified OEMs must comply with CDD guidelines and pass CTS, yet we observe partial non-compliance. This section focuses only on Google Play-certified devices [60], meaning all examined models have passed CTS at some point. Despite this certification, we find that CDD compliance does not ensure secure TLS implementations. Table 7 describes TLS stack-related CDD compliance across Android versions 9-14. Since these policies have remained unchanged, any non-compliance we observe can be attributed to OEM choices rather than CDD variations. This supports our argument that while CDD enforces compatibility, it does not prevent security-degrading customizations. Our analysis shows that 85% of OEMs fail to properly maintain system trust store, at least 9% of OEMs provide inconsistent versioning information (e.g., popular devices like Redmi C55 and OnePlus 7 Pro) and 27% of OEMs violate blocklisting and public/private API restrictions (e.g., Xiaomi Nitrogen), introducing inconsistencies even in Google-certified devices.

Google's own research confirms that passing CTS does not guarantee full CDD compliance [76], as CTS primarily ensures compatibility rather than enforcing strict security policies. As a result, OEMs can pass CTS while still introducing TLS-related deviations. While project Treble and the accompanying Vendor Test Suite (VTS) [19] improved API compliance and reduced fragmentation, they do not address critical security issues such as JCA modifications, trust store inconsistencies, and protocol-level deviations. These gaps necessitate stricter compliance enforcement and increased transparency.

7.4. Mitigation Techniques and Transparency

We note that security-degrading customizations have declined since Android 12, likely due to Treble's enforcement of API compliance. However, TLS inconsistencies persist. For certified OEMs, extending the CDD validation to explicitly test for secure TLS configurations could help mitigate these inconsistencies. Strengthening certificate validation, enforcing TLS version deprecation, and ensuring compliance with modern cryptographic standards would ensure a baseline security requirement. Expanding CTS tests—shown feasible by our work—to detect functional integrity and native dependencies could further reduce fragmentation while increasing security guarantees.

For non-certified OEMs, where Google lacks authority, stricter regulatory controls could enforce baseline security standards. For example, the 2024 EU Cyber Resilience Act (CRA) presents mandatory requirements for connected devices, ensuring vulnerability management, timely updates, and supply chain security, thereby ultimately reducing fragmentation [7]. Expanding such frameworks to include specific requirements like trust store maintenance and cryptographic standards would further strengthen the security posture for devices entering regulated markets.

Beyond these compliance enforcements, enhanced transparency on OEM customizations would allow developers and users to mitigate security risks at both app and device levels. Public findings—such as this study—could help developers implement countermeasures like certificate pinning, disabling weak TLS versions, and enforcing explicit endpoint validation. Expanding tools like FirmwareScanner to assess TLS security posture—highlighting outdated cryptographic implementations, weak ciphers, or insecure CAs—would increase user awareness of OEM security practices. Addressing TLS fragmentation requires standardization, stricter compliance enforcement, and increased transparency to ensure consistent security across the Android ecosystem.

8. Related work

Prior work analyzed the security risks associated with the lack of control over OEM customizations, from the presence of privacy-intrusive SDKs embedded in preloaded apps and malware to insecure open ports and malicious host files [34, 58, 59, 62, 65, 71, 91, 100]. Elsabagh *et al.* highlighted the lack of transparency in OEM's supply chains by showing evidence of privilege-escalation in preloaded apps, enabling unauthorized code execution and PII access [51]. Gamba *et al.* revealed privacy and security threats on preloaded Android apps [59], while a large-scale study by Hou *et al.* found widespread patch delays in 31.4% of firmware images and vulnerabilities across 8,325 firmware images from 153 OEMs [62].

Despite extensive research on the system and app level, only a few studies examine OEM modifications at the TLS stack level. El-Rewini *et al.* identified access control vulnerabilities in residual APIs, making devices susceptible to severe threats such as DoS attacks [50]. Possemato *et al.* analyzed customized Android binaries against Google's CDD, revealing loopholes within the Google compliance test but did not study their effects on secure communication [84]. Other studies identified security weaknesses at the kernel level, such as unprivileged apps gaining unauthorized access to device sensors [100, 101]. Mayrhofer *et al.* provided an overview of the Android security model, partially discussing CTS coverage but without addressing TLS considerations [75].

Lee and Wallach found vulnerabilities in BoringSSL's codebase but didn't analyze the security threats of OEMs' customizations [70]. Two studies analyzed OEMs' customizations of device-trusted root stores. Vallina-Rodriguez *et al.* showed in 2014 how opaque CAs linked to mobile operators or government agencies compromised root stores until Android 4.4 [94]. They discussed how these practices disrupt the audited root store model in AOSP and revealed TLS interception via HTTPS proxies. Ma *et al.* found that OEMs delayed removing high-severity CA certificates by over a year [73].

Our study expands on prior work by analyzing OEM modifications across all stack components, with a focus on their impact on secure communication. While previous research extensively analyzed system-level customizations such as kernel security, SE Linux policies, and application-layer risks, the impact of OEM modifications on the TLS protocol stack remains largely unexplored. Our research addresses this gap by providing empirical evidence of how these modifications introduce real-world security risks. Hence, our study holds OEMs accountable by providing large-scale empirical data on TLS deviations and offers developers actionable insights to mitigate these risks through stricter security controls at the app level.

9. Limitations and Future Work

Our data collection and black-box approach may limit the scope and depth of our findings. Yet, this method is enough to show concerning practices in OEM's protocol stack customizations and the lack of control over the supply chain. While our dataset spans from Android 9 to 15, newer versions like 15 are under-represented due to their limited deployment: as of now, there is just one stable release of Android 15. Furthermore, as outlined in §4.2, baselines were established using a best-effort approach, but this could introduce errors from misreported system build data by OEMs. We also encountered difficulties in decompiling core components (see Table 8) from several OEM devices due to corrupted files, use of obfuscation, and missing dependencies. The analysis of shared objects (.so) files was restricted to symbol trees, strings, and source file names as we lack official documentation for vendor-specific APIs. Similarly, as the contextualization of the results requires manual inspection, our analysis focuses on those methods more likely to influence apps' security according to OpenJDK documentation. The analysis in §6 highlights examples of API usage impacted by customizations found in §5, but it does not provide a systematic study of app dependencies. Finally, while certain CDD policies cover aspects related to native libraries, we left them out of scope for this study.

10. Conclusion

This paper presents the first large-scale study of OEM customizations in the Android TLS protocol stack and their impact on app security and API consistency. Our analysis reveals that critical modifications mainly affect JCA and root CA stores, while JSSE and JCE class signatures remain largely intact due to Google's CTS checks. We show that OEMs often lag on AOSP updates, which negatively affects apps' default network security and forces developers to make cautious use of TLS APIs.

While Google's CDD guidelines and CTS checks may deter OEM customizations to some extent –particularly OEMs with an interest in having their devices Google-certified–, TLS stack discrepancies remain, raising compliance and security concerns. We discussed potential root causes and mitigations, motivating the need for raising developer awareness, stricter supply-chain controls and patching processes, and more effective certification tests and compliance guidelines.

Acknowledgments

We thank the reviewers and the shepherd for their valuable feedback and suggestions for improving our paper. We also thank Suraj K. Suresh (University of California, Santa Barbara), Aniketh Girish, and Nipuna Weerasekara (IMDEA Networks Institute, Spain) for their help in the firmware analysis process. This work was motivated by discussions at the Dagstuhl seminar "EU Cyber Resilience Act: Socio-Technical and Research Challenges" (24112) in 2024. This research was partially supported by the Spanish AEI grants PID2022-143304OB-I00 (PARASITE) and PID2022-140126OB-I00 (CYCAD), both funded by MCIN/AEI/10.13039/501100011033/ and by the ERDF, EU. It was also partially supported by the Spanish National Cybersecurity Institute (INCIBE) under Proyectos Estratégicos de Ciberseguridad – CIBERSEGURIDAD EINA UNIZAR under the Recovery, Transformation and Resilience Plan funds, financed by the European Union (Next Generation). Vinuri Bandara's work has been funded by Comunidad de Madrid predoctoral grant PIPF-2023/COM-31195. Prof. N. Vallina-Rodriguez was appointed as 2019 Ramon y Cajal fellow (RYC2020-030316-I) funded by MCIN/AEI/10.13039/501100011033 and ESF Investing in your future. We used OpenAI's ChatGPT to improve the grammar, clarity, and coherence of the paper [79].

References

[1] "About knox - samsung," https://www.samsungknox.com/es-419/about-knox, [Accessed 15-10-2024].

[2] "ALPACA Attack — alpaca-attack.com," https://alpaca-attack.com/, [Accessed 10-10-2024].

[3] "Android 15 Compatibility Definition — Android Open Source Project — source.android.com,"

https://source.android.com/docs/compatibility/15/android-15-cdd#322_build_parameters, [Accessed 20-10-2024].
[4] "Behavior changes: all apps — Android Developers — developer.android.com," https://developer.android.com/about/versions/10/behavior-changes-all#tls-1.3, [Accessed 01-10-2024].
[5] "Broken or risky cryptographic algorithm — Security — Android Developers — developer.android.com," https://developer.android.com/privacy-and-security/risks/broken-cryptographic-algorithm, [Accessed 20-10-2024].
[6] "Cryptography — Security —Android Developers — developer.android.com," https://developer.android.com/privacy-and-security/cryptography#bc-algorithms, [Accessed 25-10-2024].
[7] "EU Cyber Resilience Act — digital-strategy.ec.europa.eu," https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act, [Accessed 21-02-2024].
[8] "ietf.org - rfc 2818," https://www.ietf.org/rfc/rfc2818.txt, [Accessed 01-10-2024].
[9] "Mobile Vendor Market Share Worldwide — Statcounter Global Stats — gs.statcounter.com," https://gs.statcounter.com/vendor-market-share/mobile/worldwide, [Accessed 21-10-2024].
[10] "Network security configuration — Security — Android Developers — developer.android.com," https://developer.android.com/privacy-and-security/security-config, [Accessed 25-10-2024].
[11] "NVD - CVE-2016-6709 — nvd.nist.gov," https://nvd.nist.gov/vuln/detail/CVE-2016-6709, [Accessed 22-10-2024].
[12] "NVD - CVE-2021-0341 — nvd.nist.gov," https://nvd.nist.gov/vuln/detail/CVE-2021-0341, [Accessed 21-10-2024].
[13] "OpenSSL Documentation — docs.openssl.org," https://docs.openssl.org/master/, [Accessed 24-10-2024].
[14] "platform/libcore - Git at Google — android.googlesource.com," https://android.googlesource.com/platform/libcore/, [Accessed 18-12-2023].
[15] "Remediation for Bad OpenSSL Versions - Google Help — support.google.com," https://support.google.com/faqs/answer/12576638?hl=en, [Accessed 25-10-2024].
[16] "Root of Trust — Fundamentals — Samsung Knox Documentation — docs.samsungknox.com," https://docs.samsungknox.com/admin/fundamentals/whitepaper/samsung-knox-for-android/core-platform-security/root-of-trust/, [Accessed 19-10-2024].
[17] "Security Developer's Guide — docs.oracle.com," https://docs.oracle.com/en/java/javase/11/security/index.html, [Accessed 10-10-2024].
[18] "Speeding up and strengthening HTTPS connections for Chrome on Android — security.googleblog.com," https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html, [Accessed 19-10-2024].
[19] "Here comes treble: A modular base for android," https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html, 2017, [Accessed 18-02-2024].
[20] "Android Dumps · GitLab — dumps.tadiphone.dev," https://dumps.tadiphone.dev/dumps, 2023, [Accessed 19-12-2023].
[21] "Firmware Scanner - Aplicaciones en Google Play — play.google.com," https://play.google.com/store/apps/details?id=org.imdea.networks.iag.preinstalleduploader&hl=es&gl=US, 2023, [Accessed 19-12-2023].
[22] "Android compatibility definition document," https://source.android.com/docs/compatibility/cdd, 2024, [Accessed 18-02-2024].
[23] "Boringssl - git at google," https://boringssl.googlesource.com/boringssl/, 2024, [Accessed 24-01-2024].
[24] "Codenames, tags, and build numbers — Android Open Source Project — source.android.com," https://source.android.com/docs/setup/about/build-numbers, 2024, [Accessed 26-10-2023].
[25] "GitHub - JesusFreke/smali: smali/baksmali — github.com," https://github.com/JesusFreke/smali, 2024, [Accessed 22-03-2024].
[26] "GitHub - testwhat/SmaliEx: A wrapper to get de-optimized dex from odex/oat/vdex. — github.com," https://github.com/testwhat/SmaliEx, 2024, [Accessed 22-03-2024].
[27] "Hex Rays - State-of-the-art binary code analysis solutions — hex-rays.com," https://hex-rays.com/ida-pro/, 2024, [Accessed 22-03-2024].
[28] "Libgcrypt," https://gnupg.org/software/libgcrypt/index.html, 2024, [Accessed 24-01-2024].
[29] "NVD - cve-2014-0160," https://nvd.nist.gov/vuln/detail/cve-2014-0160, 2024, [Accessed 24-01-2024].
[30] "platform/external/conscrypt - Git at Google," https://android.googlesource.com/platform/external/conscrypt/, 2024, [Accessed 27-01-2024].
[31] "platform/external/okhttp - Git at Google," https://android.googlesource.com/platform/external/okhttp/, 2024, [Accessed 27-01-2024].
[32] "Retrofit," https://square.github.io/retrofit/, 2024, [Accessed 27-01-2024].
[33] "Volley," https://google.github.io/volley/, 2024, [Accessed 27-01-2024].
[34] Y. Aafer, X. Zhang, and W. Du, "Harvesting inconsistent security configurations in custom android {ROMs} via differential analysis," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1153–1168.
[35] S. Almanee, A. Ünal, M. Payer, and J. Garcia, "Too quiet in the library: An empirical study of security updates in android apps' native code. in 2021 ieee/acm 43rd international conference on software engineering (icse)," *Los Alamitos, CA, USA*, pp. 1347–1359, 2021.
[36] Android, "Android Open Source Project - Conscrypt," https://source.android.com/docs/core/ota/modular-system/conscrypt, accessed: 2024-06-25.

[37] Android, "Documentation for Importing OpenJDK Files into libcore," https://android.googlesource.com/platform/libcore/+/HEAD/tools/expected_upstream/README.md, 2024, accessed: 2024-03-28.

[38] Android Developers, "Android api reference," https://developer.android.com/reference, 2024, accessed: 2024-03-28.

[39] Android Open Source Project, "Codenames, tags, and build numbers," https://source.android.com/docs/setup/reference/build-numbers#source-code-tags-and-builds, accessed: 2024-03-28.

[40] ARM developer, "ARM Application Binary Interface," https://developer.arm.com/Architectures/ApplicationBinaryInterface, 2024.

[41] BetaWiki, "Android 11 build rkq1.200512.002," https://betawiki.net/wiki/Android_11_build_RKQ1.200512.002, 2024, accessed: 2024-03-28.

[42] E. Blázquez, S. Pastrana, Á. Feal, J. Gamba, P. Kotzias, N. Vallina-Rodriguez, and J. Tapiador, "Trouble over-the-air: An analysis of fota apps in the android ecosystem," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1606–1622.

[43] Bouncy Castle, "Bouncy castle fips faq," http://git.bouncycastle.org/fips_faq.html, accessed: 2024-06-25.

[44] M. Caswell and O. Project, "Fix ocsp status request extension unbounded memory growth," https://github.com/openssl/openssl/commit/2c0d295e26306e15a92eb23a84a1802005c1c137, 2016, accessed: 2024-03-28.

[45] CVE Details, "Vulnerabilities in gnupg libgcrypt," https://www.cvedetails.com/vulnerability-list/vendor_id-4711/product_id-25777/Gnupg-Libgcrypt.html, 2024, accessed: 2024-03-28.

[46] Cybersecurity and Infrastructure Security Agency (CISA), "Ssl 3.0 protocol vulnerability and poodle attack," https://www.cisa.gov/news-events/alerts/2014/10/17/ssl-30-protocol-vulnerability-and-poodle-attack, 2014, accessed: 2024-03-28.

[47] T. Dai, H. Shulman, and M. Waidner, "Let's downgrade let's encrypt," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1421–1440. [Online]. Available: https://doi.org/10.1145/3460120.3484815

[48] A. developer, "GNU toolchain," https://developer.arm.com/Architectures/ABI, 2024.

[49] Z. Dong, Y. Zhao, T. Liu, C. Wang, G. Xu, G. Xu, and H. Wang, "Same app, different behaviors: Uncovering device-specific behaviors in android apps," *arXiv preprint arXiv:2406.09807*, 2024.

[50] Z. El-Rewini and Y. Aafer, "Dissecting residual apis in custom android roms," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1598–1611.

[51] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "{FIRMSCOPE}: Automatic uncovering of {Privilege-Escalation} vulnerabilities in {Pre-Installed} apps in android firmware," in *29th USENIX security symposium (USENIX Security 20)*, 2020, pp. 2379–2396.

[52] facebookincubator, "Fizz: C++14 implementation of the tls-1.3 standard," https://github.com/facebookincubator/fizz, 2024, accessed: 2024-03-28.

[53] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking ssl development in an appified world," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 49–60. [Online]. Available: https://doi.org/10.1145/2508859.2516655

[54] Á. Feal, P. Calciati, N. Vallina-Rodriguez, C. Troncoso, and A. Gorla, "Angel or devil? a privacy study of mobile parental control apps," *Proceedings on Privacy Enhancing Technologies*, 2020.

[55] O. Foundation, "Openssl," https://www.openssl.org/, 2024, [Accessed 24-01-2024].

[56] J. Friess, H. Schulmann, and M. Waidner, "Revocation speedrun: How the webpki copes with fraudulent certificates," *Proc. ACM Netw.*, vol. 1, no. CoNEXT3, Nov. 2023. [Online]. Available: https://doi.org/10.1145/3629148

[57] J. Gajrani, U. Agarwal, V. Laxmi, B. Bezawada, M. S. Gaur, M. Tripathi, and A. Zemmari, "Espydroid+: Precise reflection analysis of android apps," *Computers Security*, vol. 90, p. 101688, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404819302251

[58] J. Gamba, Á. Feal, E. Blazquez, V. Bandara, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "Mules and permission laundering in android: Dissecting custom permissions in the wild," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–18, 2023.

[59] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1039–1055.

[60] Google, "Supported devices - Google Play Help - Google Support — storage.googleapis.com," https://storage.googleapis.com/play_public/supported_devices.html, [Accessed 18-02-2025].

[61] Google, "Google android enterprise security whitepaper 2018," https://source.android.com/static/docs/security/overview/reports/Google_Android_Enterprise_Security_Whitepaper_2018.pdf, 2018, accessed: 2024-03-28.

[62] Q. Hou, W. Diao, Y. Wang, C. Mao, L. Ying, S. Liu, X. Liu, Y. Li, S. Guo, M. Nie, and H. Duan, "Can we trust the phone vendors? comprehensive security measurements on the android firmware ecosystem," *IEEE Transactions on Software Engineering*, vol. 49, no. 07, pp. 3901–3921, jul 2023.

[63] G. B. Hunters, "Google bug hunters report," https://bughunters.google.com/report, accessed: 2024-07-05.

[64] K. Hynek, D. Vekshin, J. Luxemburk, T. Cejka, and

A. Wasicek, "Summary of dns over https abuse," *IEEE Access*, vol. 10, pp. 54 668–54 680, 2022.

[65] Y. J. Jia, Q. A. Chen, Y. Lin, C. Kong, and Z. M. Mao, "Open doors for bob and mallory: Open port usage in android apps and security implications," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 190–203.

[66] K. R. Jones, T.-F. Yen, S. C. Sundaramurthy, and A. G. Bardas, "Deploying android security updates: an extensive study involving manufacturers, carriers, and end users," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 551–567.

[67] J. Koret, "Diaphora," https://github.com/joxeankoret/diaphora.

[68] P. Kotzias, A. Razaghpanah, J. Amann, K. G. Paterson, N. Vallina-Rodriguez, and J. Caballero, "Coming of age: A longitudinal study of tls deployment," in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 415–428. [Online]. Available: https://doi.org/10.1145/3278532.3278568

[69] J. Larisch, W. Aqeel, T. Chung, E. Kohler, D. Levin, B. M. Maggs, B. Parno, and C. Wilson, "No root store left behind," in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, 2023, pp. 295–301.

[70] J. Lee and D. S. Wallach, "Removing secrets from android's tls," in *NDSS*, 2018.

[71] D. 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*. Springer, 2021, pp. 231–251.

[72] Y. Lindell, "RFC 8452: AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption — datatracker.ietf.org," https://datatracker.ietf.org/doc/html/rfc8452, [Accessed 19-10-2024].

[73] Z. Ma, J. Austgen, J. Mason, Z. Durumeric, and M. Bailey, "Tracing your roots: exploring the tls trust anchor ecosystem," in *Proceedings of the 21st ACM Internet Measurement Conference*, 2021, pp. 179–194.

[74] A. N. Malpani, "RFC 2560: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP — datatracker.ietf.org," https://datatracker.ietf.org/doc/html/rfc2560, 1999, [Accessed 30-04-2024].

[75] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The android platform security model," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 3, pp. 1–35, 2021.

[76] R. Mayrhofer, J. Vander Stoep, C. Brubaker, D. Hackborn, B. Bonne, G. S. Tuncay, R. Piqueras Jover, and M. A. Specter, "The android platform security model (2023)," *arXiv preprint arXiv:1904.05572*, 2024.

[77] h. . https://developer.android.com/about/versions/12/behavior-changes-all#bouncy-castle. n. . A. ndroid, year = 2024, "Behavior changes: all apps - Android 12."

[78] M. Oltrogge, N. Huaman, S. Amft, Y. Acar, M. Backes, and S. Fahl, "Why eve and mallory still love android: Revisiting {TLS}({In) Security} in android applications," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 4347–4364.

[79] OpenAI, "Chatgpt, march 2024 version," https://chat.openai.com, 2024, accessed: April 1, 2025.

[80] OpenSSL Foundation, "/docs/-man3.0/man7/crypto.html — openssl.org," https://www.openssl.org/docs/man3.0/man7/crypto.html, 2024, [Accessed 22-03-2024].

[81] Oracle Co-operation, "Openjdk," https://openjdk.org/, 2024, [Accessed 23-01-2024].

[82] OWASP Foundation, "Insecure randomness," https://owasp.org/www-community/vulnerabilities/Insecure_Randomness, accessed: 2024-03-28.

[83] S. Pletinckx, T. Nguyen, T. Fiebig, C. Kruegel, and G. Vigna, "Certifiably vulnerable: Using certificate transparency logs for target reconnaissance," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, jul 2023.

[84] A. Possemato, S. Aonzo, D. Balzarotti, and Y. Fratantonio, "Trust, but verify: A longitudinal analysis of android oem compliance and customization," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 87–102.

[85] A. O. S. Project, "Compatibility test suite (cts)," https://source.android.com/docs/compatibility/cts, 2024, accessed: 2024-03-28.

[86] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill, "Studying tls usage in android apps," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, 2017, pp. 350–362.

[87] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, P. Gill *et al.*, "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem," in *The 25th annual network and distributed system security symposium (NDSS 2018)*, 2018.

[88] E. Rescorla, K. Oku, N. Sullivan, and C. A. Wood, "TLS Encrypted Client Hello," Internet Engineering Task Force, Internet-Draft draft-ietf-tls-esni-23, Feb. 2025, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tls-esni/23/

[89] I. Square, "Okhttp: Square's meticulous http client for the jvm, android, and graalvm," https://github.com/square/okhttp, accessed: 2024-06-25.

[90] Statcounter, "Android version market share worldwide," https://gs.statcounter.com/android-version-market-share, accessed: 2024-06-25.

[91] T. Sutter and B. Tellenbach, "Firmwaredroid: Towards automated static analysis of pre-installed android apps," in *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2023, pp. 12–22.

[92] The Chromium Projects, "Chrome root program policy, version 1.5," https://www.chromium.org/Home/chromium-security/root-ca-policy/, 2024,

last updated: 2024-01-16, Accessed: 2024-03-28.
[93] The Washinton Post, "Web browsers drop mysterious company with ties to u.s. military contractor," https://www.washingtonpost.com/technology/2022/11/30/trustcor-internet-authority-mozilla/, 2022, accessed: 2024-04-20.
[94] N. Vallina-Rodriguez, J. Amann, C. Kreibich, N. Weaver, and V. Paxson, "A tangled mass: The android root certificate stores," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 141–148.
[95] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 226–237.
[96] Wired, "Independent iranian hacker claims responsibility for comodo hack," https://www.wired.com/2011/03/comodo-hack/, 2011, last updated: 2011-03-28, Accessed: 2024-03-29.
[97] H. Xiaomi, "Hackerone report to xiaomi," https://hackerone.com/xiaomi, accessed: 2024-10-17.
[98] K. S. YIM, I. Malchev, A. Hsieh, and D. Burke, "Treble: Fast software updates by creating an equilibrium in an active software ecosystem of globally distributed stakeholders," *ACM Trans. Embed. Comput. Syst.*, vol. 18, pp. 104:1–104:23, 2019.
[99] Zebra Technologies, "Android mobile dod stig validation," https://www.zebra.com/content/dam/zebra_dam/en/brief/application/government-brief-application-android-mobile-dod-stig-validation-en-us.pdf, 2024, accessed: 2024-03-28.
[100] M. Zheng, M. Sun, and J. C. Lui, "Droidray: a security evaluation system for customized android firmwares," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 471–482.
[101] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 409–423.

Appendix A.
Data Availability (Open Science)

Code availability: The code and associated scripts used in this research are publicly available on GitHub and can be accessed at: https://github.com/OEM-customization/. This repository includes the full automation script to clone and build Android images, as well as the diffing pipeline for comparing OEM firmware images against AOSP baselines. Additionally, we release the complete list of customizations observed for independent verification, including detailed information such as vendor, model, fingerprint, baseline, component, class name, and customization. These results are stored in the `results/` folder within the repository for public access.

Data availability: The dataset used in this study consists of sample Android dumps (images), and links to these images are provided in the repository under the `firmware_images/sample_oem.txt` file. Users can clone more Android dumps from the Android dumps website [20].

Reproducibility materials: We provide sample AOSP images for major Android releases in the `AOSP_BUILDS/` folder within the repository. A detailed automated diffing pipeline along with an AOSP image building script is included, allowing users to replicate our results by comparing OEM firmware images to these AOSP baselines.

System requirements and environment setup: To successfully clone, build, and run the pipeline, ensure your machine meets the following requirements:

- Disk Space: At least 400 GB of free disk space (250 GB for checking out images + 150 GB for building them).
- Software Packages: The required packages from the Android Open Source Project must be installed, as well as these additional Python libraries: pandas, numpy, ssdeep, colorama, pyOpenSSL.

Running the pipeline: After setting up your environment and ensuring you have sufficient disk space:

1) Download OEM Firmware Images: Navigate to the `firmware_images/` directory and clone the repositories listed in `sample_oem.txt`.
2) Prepare AOSP Images: We provide 5 baseline AOSP images in the `AOSP_BUILDS/` folder. If your target image differs, the pipeline will automatically build the necessary AOSP version.
3) Execute the Pipeline: Run the `main.py` script, follow the on-screen prompts, and proceed with the diffing process. Review Results: All outputs are saved in the `diffing_results/` folder. To determine the role of missing functions (AOSP-only functions), refer to the Android documentation on the APIs [38]. For detecting the role of OEM-specific methods/-logic, manually review and establish the usage of the methods. Refer to Appendix B.3 to understand how compiler-introduced methods are filtered.

Appendix B.
Supplementary Material

This section provides complementary material about component extraction and manual verification processes, with various examples.

B.1. TLS Stack Component Identification

In § 4, we discuss the methodology to extract TLS components. Table 8 lists the various paths considered.

B.2. Baseline Identification

As explained in § 4.2, we detected cases where the `BUILD_ID` did not directly correspond to a known AOSP release or could not be found online. Table 9 exemplifies several identified inconsistencies for reference.

B.3. Diffing result contextualization guidelines.

When conducting automated and manual analysis of function and logic-level diffing for both Java and C++ libraries, we follow these structured guidelines:

1) For Java libraries:
   - Identifying missing AOSP functions: When OEMs exclude functions available in AOSP, we cross-reference the missing function with the official Android sources for JSSE [14], Conscrypt [30] and OkHttp [31]. Additionally, Android API documentation [38] is used to understand the expected behavior and purpose of these functions.
   - Analyzing logic diffs: For logic diffs, we closely inspect the custom logic introduced by the OEMs, identifying any extra checks, or branches that differ from the standard AOSP implementation. We document how these changes may affect app behavior or security. We note that this requires the ability to understand the Smali-level representation of the code.
   - Understanding OEM-implemented functions: For custom Java functions introduced by OEMs, we analyze the Smali representations of the corresponding class and function code. We study the intended utility of these new functions and trace how they are invoked by other components within the framework to determine their role and potential impact.
2) For Native libraries (C++)
   - BoringSSL function analysis: Due to the inability to extract source code from shared objects, we depend on the documented behaviors of BoringSSL [23] and OpenSSL [13] to study the impact of OEM customizations. We begin by examining the symbol tree. This allows us to detect added or missing functions.
   - Understanding missing cryptographic functions: When the pipeline detects a missing function, we study its usage in the AOSP to understand the function's role and assess the impact of its absence in the OEM customization. For added functions,

TABLE 8: Default and probable locations for the networking libraries, packages, and core components.

File Type	Default Path	Description
boot.oat	/system/framework/arm(64)/boot.oat	Includes the core-oj components including javax.net, javax.crypto, etc. which are forked 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 forked by OpenJDK
Native libraries	*/lib */lib64	Includes the compiled libraries built during runtime including the JCA providers
Conscrypt	/framework/arm(64)/boot-conscrypt.oat /framework/conscrypt.odex(.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(64)/boot-okhttp.oat /framework/okhttp.odex(.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(64)/boot-bouncycastle.oat /framework/bouncycastle.odex(.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
System CA certificates	/etc/security/cacerts	Includes the root store certificates of each device

TABLE 9: Examples of inconsistencies between `build.prop` and build metadata for selected devices.

OEM	Model	Build ID	Version reported by OEM (API level)	API level defined in build.prop	Last build date defined in build.prop	Defined Baseline
Mito	v152_jxd_2h2	LMY47D	9 (28)	22	03/29/2016	android-9.0.0_r1
Asus	X00TD	qkq1	9 (28)	31	01/19/2022	android-9.0.0_r1
Motorola	kiev_retail	rzks31.q3-45-16-8-3	11 (30)	32	07/11/2022	android-11.0.0_r1
Samsung	graceltexx	mmb29k	9 (28)	24	09/10/2020	android-9.0.0_r1
Alps	f202_f22_p10	mra58k	10 (29)	23	08/24/2020	android-10.0.0_r1
Samsung	j6primelte	QP1A.190711.020	10 (29)	29	07/02/2020	android-10.0.0_r2
Redmi	matisse	sp1a.210812.016	12 (31)	31	11/04/2022	android-12.0.0_r3

we check if the added functions match with any OpenSSL functions that don't exist in BoringSSL.

Figure 7 exemplifies how we differentiate between obfuscation and legitimate functional changes, as explained in § 4.3.

```
.class public final synthetic Lsun/nio/ch/-$$Lambda$ThreadPool$N88rfRTSpCtnK5fgJO
      -WA6OwVQM;
.source "lambda"

# interfaces
.implements Ljava/util/concurrent/ThreadFactory;

# static fields
.field public static final synthetic INSTANCE:Lsun/nio/ch/-
      $$Lambda$ThreadPool$N88rfRTSpCtnK5fgJO-WA6OwVQM;

# direct methods
.method static synthetic constructor <clinit>()V
    .registers 1

    new-instance v0, Lsun/nio/ch/-$$Lambda$ThreadPool$N88rfRTSpCtnK5fgJO-WA6OwVQM
          ;

    invoke-direct {v0}, Lsun/nio/ch/-$$Lambda$ThreadPool$N88rfRTSpCtnK5fgJO-
          WA6OwVQM;-><init>()V

    sput-object v0, Lsun/nio/ch/-$$Lambda$ThreadPool$N88rfRTSpCtnK5fgJO-WA6OwVQM
          ;->INSTANCE:Lsun/nio/ch/-$$Lambda$ThreadPool$N88rfRTSpCtnK5fgJO-
          WA6OwVQM;

    #disallowed odex opcode
    #return-void-no-barrier
    nop
.end method

.method private synthetic constructor <init>()V
    .registers 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
    return-void
.end method
```

Figure 7: Example of synthetic and obfuscation classes introduced by compilation artifacts. We filter out functions with the synthetic modifier.

Appendix C.
Supplementary code snippets

```
// Class: java.net.NetworkInterface
public byte[] getHardwareAddress() throws SocketException {
    NetworkInterface ni = getByName(this.name);
    if (ni == null) throw new SocketException("Interface missing");
    if (ni.hardwareAddr == null && !"lo".equals(this.name)
        && !Compatibility.isChangeEnabled(RETURN_NULL_HARDWARE_ADDRESS)) {
        return DEFAULT_MAC_ADDRESS.clone();
    }
    return shouldProtectMac(this.name) ? DEFAULT_MAC_ADDRESS.clone() : ni.
          hardwareAddr;
}

private boolean shouldProtectMac(String iface) {
    if (!"wlan0".equals(iface)) return false;
    try {
        synchronized (NetworkInterface.class) {
            if (checkMacPermissionMethod == null) {
                Class<?> cls = Class.forName("com.vivo.services.security.
                      client.VivoPermissionManager");
                checkMacPermissionMethod = cls.getMethod("
                      checkCallerMacPermission", String.class);
            }
        }
        return !((Boolean) checkMacPermissionMethod.invoke(null, iface));
    } catch (Exception e) { return false; }
}
// Class: com.vivo.framework.security.VivoPermissionManager
public boolean checkCallerMacPermission(String iface) {
    return isOverSeas() || isSystemAppCalling(this.mContext)
          || SystemProperties.getBoolean("persist.vivo.wifimacrandom", true);
}
// Class: com.bbk.account.base.utils.AccountSystemProperties
private AccountSystemProperties() {
    this.mCountryCode = getSystemProperties("ro.product.country.region", "N");
    if ("N".equals(this.mCountryCode))
        this.mCountryCode = getSystemProperties("ro.product.customize.bbk", "N
              ");
    this.mIsOverseas = "CN".equals(getSystemProperties(SYSTEM_KEY_OVERSEAS, "
          no"));
    if (!this.mIsOverseas) this.mCountryCode = "CN";
}
public String getCountryCode() { return this.mCountryCode; }
public boolean isOverseas() { return this.mIsOverseas; }
```

Figure 8: Java representation of Vivo MAC address protection with region-specific, permission-specific conditions.

Figures 13, 14, 15, 16, 17 an18 show code-snippets of Android apps making use of missing or customized TLS functions by OEMs.

```
package com.android.server.pm;

public class CompatibilityHelper extends com.oppo.RomUpdateHelper {

public void customizeNativeLibrariesIfNeeded(PackageParser.Package pkg) {
        File[] listFiles;
        ArrayList tmpList = new ArrayList();
        boolean bOpenssl = false;
        if (isInWhiteList(23, pkg.packageName)) {
            tmpList.add("openssl");
        }
        File dir = new File(pkg.applicationInfo.nativeLibraryDir);
        if (dir.isDirectory()) {
            for (File tmp : dir.listFiles()) {
                String libName = tmp.getName();
                if (libName != null) {
                    if (!bOpenssl && isInWhiteList(24, libName)) {
                        if (!tmpList.contains("openssl")) {
                            tmpList.add("openssl");
                        }
                    } else if ("libssl.so".equals(libName) || "libcrypto.so".
                          equals(libName)) {
                        bOpenssl = true;
                    }
                    if ("libdexinterpret.so".equals(libName) && isInWhiteList(28,
                          pkg.packageName)) {
                        tmpList.add("atlas");
                    }
                }
            }
            if (bOpenssl || isInWhiteList(FORCE_DISABLE_OPENSSL, pkg.packageName)
                  ) {
                tmpList.remove("openssl");
            }
        }
        if (tmpList.size() > 0) {
            pkg.applicationInfo.specialNativeLibraryDirs = (String[]) tmpList.
                  toArray(new String[tmpList.size()]);
        }
        if (isInWhiteList(FORCE_NEED_SPECIAL_LIBRARIES_IN, pkg.packageName)) {
            boolean is64Bit = VMRuntime.is64BitInstructionSet(InstructionSets.
                  getPrimaryInstructionSet(pkg.applicationInfo));
            String openssllibraryDir = is64Bit ? "/vendor/lib64/openssl" : "/
                  vendor/lib/openssl";
            pkg.applicationInfo.nativeLibraryDir = openssllibraryDir + File.
                  pathSeparator + pkg.applicationInfo.nativeLibraryDir;
        }
}
```

Figure 9: Oppo usage of OpenSSL for compatibility. The CompatibilityHelper class uses outdated OpenSSL versions from the years 2013 and 2014 with known vulnerabilities.

```
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Scanned Source License libgcrypt-1.8.3/LICENSES:

Additional license notices for Libgcrypt. -*- org -*-

This file contains the copying permission notices for various files in the
      Libgcrypt distribution which are not covered by the GNU Lesser General
      Public License (LGPL) or the GNU General Public License (GPL).

These notices all require that a copy of the notice be included in the
      accompanying documentation and be distributed with binary distributions of
       the code, so be sure to include this file along with any binary
      distributions derived from the GNU C Library.

* BSD_3Clause
  For files:
  - cipher/sha256-avx-amd64.S
  - cipher/sha256-avx2-bmi2-amd64.S
  - cipher/sha256-ssse3-amd64.S
  - cipher/sha512-avx-amd64.S
  - cipher/sha512-avx2-bmi2-amd64.S
  - cipher/sha512-ssse3-amd64.S

#+begin_quote
  Copyright (c) 2012, Intel Corporation All rights reserved.

  Redistribution and use in source and binary forms, with or without modification
        , are permitted provided that the following conditions are met:

  * Redistributions of source code must retain the above copyright notice, this
        list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above copyright notice,
        this list of conditions and the following disclaimer in the
        documentation and/or other materials provided with the distribution.

  * Neither the name of the Intel Corporation nor the names of its contributors
        may be used to endorse or promote products derived from this software
        without specific prior written permission.


  THIS SOFTWARE IS PROVIDED BY INTEL CORPORATION "AS IS" AND ANY EXPRESS OR
        IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
        WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
        DISCLAIMED. IN NO EVENT SHALL INTEL CORPORATION OR CONTRIBUTORS BE
        LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
        CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
        SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
         INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
         CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
         ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
        THE POSSIBILITY OF SUCH DAMAGE.
#+end_quote

  For files:
  - random/jitterentropy-base.c
  - random/jitterentropy.h
  - random/rndjent.c (plus common Libgcrypt copyright holders)

#+begin_quote
 * Copyright Stephan Mueller &lt;smueller@chronox.de&gt;, 2013
```

Figure 10: Libgcrypt usage in Google Home implementations by LGE and HTC, incorporated for its SHA-256 and SHA-512 functionalities.

```
// OpenSSL engine usage for Samsung smart cards

package com.sec.enterprise.jce.provider.pkcs11;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.security.InvalidKeyException;
import java.security.PrivateKey;
public final class OpenSSLEnginePrivateKeyHelper {
    private static final int SECPKCS11_ENGINE_CAC_CUSTOM_CTX = 2;
    private static final int SECPKCS11_ENGINE_CCM_CUSTOM_CTX = 1;
    private static final String TAG = "OpenSSLEnginePrivateKeyHelper";
    private static native long ENGINE_load_private_key(int i, String str);

    static {
        System.loadLibrary("secpkcs11_engine.secsmartcard.samsung");
    }

    private OpenSSLEnginePrivateKeyHelper() {
    }
    public static PrivateKey ccmGetPrivateKeyById(String id) throws
           InvalidKeyException {
        return getPrivateKeyById(1, id);
    }
    public static PrivateKey cacGetPrivateKeyById(String id) throws
           InvalidKeyException {
        return getPrivateKeyById(2, id);
    }
    private static PrivateKey getPrivateKeyById(int handle, String id) throws
           InvalidKeyException {
        if (id == null) {
            throw new NullPointerException("id == null");
        }
        long keyRef = ENGINE_load_private_key(handle, id);
        if (keyRef == 0) {
            return null;
        }
        try {
            Class c = Class.forName("com.android.org.conscrypt.OpenSSLKey");
            Constructor a = c.getDeclaredConstructor(Long.TYPE, Boolean.TYPE,
                    Boolean.TYPE);
            a.setAccessible(true);
            Object instance = a.newInstance(Long.valueOf(keyRef), false, true);
            Method getPrivateKey = c.getDeclaredMethod("getPrivateKey", null);
            getPrivateKey.setAccessible(true);
            PrivateKey pkey = (PrivateKey) getPrivateKey.invoke(instance, null);
            return pkey;
        } catch (Exception e) {
            return null;
        }
    }
}

// Smart card helper class in Samsung devices
package com.sec.smartcard.client;
import com.sec.smartcard.pkcs11.Pkcs11;

public class SmartCardHelper {
    public static final String OWN_CRYPTOKI_LIBRARYNAME = "
           libSamsungPkcs11Wrapper.secsmartcard.samsung.so";
    static { System.loadLibrary("secpkcs11_engine.secsmartcard.samsung"); }

    private ICryptoServiceListener mCryptoServiceListener = new
           ICryptoServiceListener.Stub() {
        @Override
        public void onSmartCardConnected(String identifier) {
            Log.i("SmartcardHelper", "onSmartCardConnected");
            synchronized (this) {
                Log.i("SmartcardHelper", "Executing synchronized block");
                if (SmartCardHelper.this.mPkcs11Loader == null) {
                    Log.i("SmartcardHelper", "Pkcs11 is null. So creating and
                           loading Pkcs11");
                    SmartCardHelper.this.mPkcs11Loader = new Pkcs11();
                    if (SmartCardHelper.this.mPkcs11Loader != null) {
                        Log.i("SmartcardHelper", "Load Pkcs11");
                        if (SmartCardHelper.this.mBoundSmartCardService != null)
                              {
                            try {
                                String socket = SmartCardHelper.this.
                                       mBoundSmartCardService.
                                       getCryptoServiceSocket();
                                SmartCardHelper.this.mPkcs11Loader.Load(
                                       SmartCardHelper.OWN_CRYPTOKI_LIBRARYNAME,
                                        SmartCardHelper.
                                              OWN_C_GETFUNCTIONLIST_METHOD,
                                              socket, false);
                            } catch (RemoteException e) {
                                e.printStackTrace();
                            }
                        }}}}
            SmartCardHelper.this.mCallback.onInitComplete();
        }
        @Override
        public void onSmartCardConnectionError(String identifier, int reasonCode)
                {
            Log.i("SmartcardHelper", "onSmartCardConnectionError" + reasonCode);
            try {
                synchronized (this) {
                    if (SmartCardHelper.this.mPkcs11Loader != null) {
                        SmartCardHelper.this.mPkcs11Loader.Unload();
                        SmartCardHelper.this.mPkcs11Loader = null;
                    }
                }
            } catch (RemoteException e) {
                e.printStackTrace();
        }}
```

Figure 11: PKCS11_engine.so usage for Samsung smart card services, is important for Samsung's TrustZone implementations for securing sensitive data.

```
\\ Advanced crypto (AC) symmetric key factory extending Java security
      KeyFactorySpi
package com.xiaomi.finddevice.common.advancedcrypto;

import java.security.Key;
import java.security.KeyFactorySpi;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.KeySpec;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;

public class ACAsymKeyFactory extends KeyFactorySpi {
    static { System.loadLibrary("advanced_crypto_jni"); }
    private static native String getPrivateKeyType(byte[] bArr);
    private static native String getPublicKeyType(byte[] bArr);
    @Override // java.security.KeyFactorySpi
    protected final PrivateKey engineGeneratePrivate(KeySpec keySpec) throws
          InvalidKeySpecException {
        ACECCPrivateKey build;
        if (keySpec instanceof PKCS8EncodedKeySpec) {
            byte[] encoded = ((PKCS8EncodedKeySpec) keySpec).getEncoded();
            if (encoded != null) {
                if (getPrivateKeyType(encoded).equals("ACECCPrivate")) {
                    try {
                        build = ACECCPrivateKey.build(encoded);
                    } catch (ACBadKeyBytesException unused) {
                        throw new RuntimeException("Should never reach here. ");
                    }
                } else {
                    build = null;
                }
                if (build != null) {
                    return build;
                }
               ...
        throw new InvalidKeySpecException("Support only PKCS8EncodedKeySpec");
    }

    @Override // java.security.KeyFactorySpi
    protected final PublicKey engineGeneratePublic(KeySpec keySpec) throws
          InvalidKeySpecException {
        ACDSAPublicKey build;
        if (keySpec instanceof X509EncodedKeySpec) {
            byte[] encoded = ((X509EncodedKeySpec) keySpec).getEncoded();
            if (encoded != null) {
                if (getPublicKeyType(encoded).equals("ACDSAPublic")) {
                    try {
                        build = ACDSAPublicKey.build(encoded);
                    } catch (ACBadKeyBytesException unused) {
                        throw new RuntimeException("Should never reach here. ");
                    }
                } else {
                    build = null;
                }
                if (build != null) {
                    return build;
                }
               ...
        throw new InvalidKeySpecException("Support only X509EncodedKeySpec");
}

\\ ACJCE implementation extending Java security provider
package com.xiaomi.finddevice.common.advancedcrypto;
import java.security.Provider;
public class ACJCEProvider extends Provider {
    public ACJCEProvider() {
        super("AC", 1.0d, "Native Advanced Crypto");
        put("KeyFactory.ACAsym", ACAsymKeyFactory.class.getName());
        put("KeyFactory.EC", ACECCKeyFactory.class.getName());
        put("KeyFactory.DSA", ACDSAKeyFactory.class.getName());
        put("Signature.sha256withECDSA", ACSha256withECDSASignature.class.getName
              ());
        put("Signature.DSA", ACDSASignature.class.getName());
    }
}

\\ ACJCE provider usage in Xiaomi's find device app.
package com.xiaomi.finddevice;
import com.xiaomi.finddevice.common.advancedcrypto.ACJCEProvider;
import java.security.Security;
public class Application extends android.app.Application {
    Security.insertProviderAt(new ACJCEProvider(), 1);
    FindDeviceKeyguardManager.init(this);
}
```

Figure 12: The libadvanced_crypto.so usage supporting MIUI devices Find device app, building on the capabilities provided by libcrypto.so.

```
public byte[] encrypt(com.microsoft.appcenter.utils.crypto.
      CryptoUtils$ICryptoFactory p1, int p2, java.security.KeyStore$Entry p3,
      byte[] p4) {
    byte[] v1_1 = this.getCipher(p1, p2);
    java.security.InvalidKeyException v2_3 = ((java.security.cert.X509Certificate
          ) ((java.security.KeyStore$PrivateKeyEntry) p3).getCertificate());
    try {
        v2_3.checkValidity();
    } catch (byte[] v1_3) {
        throw new java.security.InvalidKeyException(v1_3);
    }
    v2_3.getPublicKey();
    v1_1.init(1, v2_3);
    return v1_1.doFinal(p4);
}
```

Figure 13: encrypt() function using JSSE's checkValidity to check the certificate validity before allowing encryption operations within the app.

```
public final com.squareup.okhttp.Response readNetworkResponse() {
    this.transport.finishRequest();
    com.squareup.okhttp.Response response = this.transport.readResponseHeaders()
        .request(this.networkRequest)
        .handshake(this.connection.getHandshake())
        .header(com.squareup.okhttp.internal.http.OkHeaders.SENT_MILLIS, Long.
              toString(this.sentRequestMillis))
        .header(com.squareup.okhttp.internal.http.OkHeaders.RECEIVED_MILLIS, Long
              .toString(System.currentTimeMillis()))
        .build();

    if (!this.forWebSocket) {
        response = response.newBuilder().body(this.transport.openResponseBody(
              response)).build();
    }
    return response;
}
```

Figure 14: OkHttp.handshake() function used to extract handshake details, the readNetworkResponse() function lacks proper exception handling in the scenario where the handshake() function is missing.

```
.method public connect(IIIIZLcom/mbridge/msdk/thrid/okhttp/Call;Lcom/mbridge/msdk
      /thrid/okhttp/EventListener;)V
    .locals 16
    .param p1, "connectTimeout"    # I
    .param p2, "readTimeout"    # I
    .param p3, "writeTimeout"    # I
    .param p4, "pingIntervalMillis"    # I
    .param p5, "connectionRetryEnabled"    # Z
    .param p6, "call"    # Lcom/mbridge/msdk/thrid/okhttp/Call;
    .param p7, "eventListener"    # Lcom/mbridge/msdk/thrid/okhttp/EventListener;
    iget-object v0, p0, Lcom/mbridge/msdk/thrid/okhttp/internal/connection/
          RealConnection;->protocol:Lcom/mbridge/msdk/thrid/okhttp/Protocol;
    if-nez v0, :cond_b
    iget-object v1, p0, Lcom/mbridge/msdk/thrid/okhttp/internal/connection/
          RealConnection;->route:Lcom/mbridge/msdk/thrid/okhttp/Route;
    invoke-virtual {v1}, Lcom/mbridge/msdk/thrid/okhttp/Route;->address()Lcom/
          mbridge/msdk/thrid/okhttp/Address;
    move-result-object v1
    invoke-virtual {v1}, Lcom/mbridge/msdk/thrid/okhttp/Address;->
          sslSocketFactory()Ljavax/net/ssl/SSLSocketFactory;
    move-result-object v1
    if-nez v1, :cond_2
    invoke-virtual {v1}, Lcom/mbridge/msdk/thrid/okhttp/Address;->url()Lcom/
          mbridge/msdk/thrid/okhttp/HttpUrl;
    move-result-object v1
    invoke-virtual {v1}, Lcom/mbridge/msdk/thrid/okhttp/HttpUrl;->host()Ljava/
          lang/String;
    move-result-object v1
    invoke-static {}, Lcom/mbridge/msdk/thrid/okhttp/internal/platform/Platform
          ;->get()Lcom/mbridge/msdk/thrid/okhttp/internal/platform/Platform;
    move-result-object v2
    invoke-virtual {v2, v1}, Lcom/mbridge/msdk/thrid/okhttp/internal/platform/
          Platform;->isCleartextTrafficPermitted(Ljava/lang/String;)Z
    move-result v2
    if-eqz v2, :cond_0
    goto :goto_0
    :cond_0
    new-instance v2, Lcom/mbridge/msdk/thrid/okhttp/internal/connection/
          RouteException;
    new-instance v3, Ljava/net/UnknownServiceException;
    new-instance v4, Ljava/lang/StringBuilder;
    invoke-direct {v4}, Ljava/lang/StringBuilder;-><init>()V
    const-string v5, "CLEARTEXT communication to "
    invoke-virtual {v4, v5}, Ljava/lang/StringBuilder;->append(Ljava/lang/String
          ;)Ljava/lang/StringBuilder;
    invoke-virtual {v4, v1}, Ljava/lang/StringBuilder;->append(Ljava/lang/String
          ;)Ljava/lang/StringBuilder;
    const-string v5, " not permitted by network security policy"
    invoke-virtual {v4, v5}, Ljava/lang/StringBuilder;->append(Ljava/lang/String
          ;)Ljava/lang/StringBuilder;
    invoke-virtual {v4}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;
    move-result-object v4
    invoke-direct {v3, v4}, Ljava/net/UnknownServiceException;-><init>(Ljava/lang
          /String;)V
    invoke-direct {v2, v3}, Lcom/mbridge/msdk/thrid/okhttp/internal/connection/
          RouteException;-><init>(Ljava/io/IOException;)V
    throw v2
    :cond_2
    move-object v12, v0
    iget-object v0, p0, Lcom/mbridge/msdk/thrid/okhttp/internal/connection/
          RealConnection;->route:Lcom/mbridge/msdk/thrid/okhttp/Route;
    invoke-virtual {v0}, Lcom/mbridge/msdk/thrid/okhttp/Route;->requiresTunnel()Z
    move-result v0
    if-eqz v0, :cond_4
    invoke-direct/range {p0 .. p7}, Lcom/mbridge/msdk/thrid/okhttp/internal/
          connection/RealConnection;->connectTunnel(IIILcom/mbridge/msdk/thrid/
          okhttp/Call;Lcom/mbridge/msdk/thrid/okhttp/EventListener;)V
    iget-object v0, p0, Lcom/mbridge/msdk/thrid/okhttp/internal/connection/
          RealConnection;->rawSocket:Ljava/net/Socket;
    if-nez v0, :cond_3
    goto :goto_3
    :cond_3
    invoke-direct {p0, p1, p2, p6, p7}, Lcom/mbridge/msdk/thrid/okhttp/internal/
          connection/RealConnection;->connectSocket(IILcom/mbridge/msdk/thrid/
          okhttp/Call;Lcom/mbridge/msdk/thrid/okhttp/EventListener;)V
    move/from16 v15, p4
    invoke-direct {p0, v11, v15, p6, p7}, Lcom/mbridge/msdk/thrid/okhttp/internal
          /connection/RealConnection;->establishProtocol(Lcom/mbridge/msdk/thrid
          /okhttp/internal/connection/ConnectionSpecSelector;ILcom/mbridge/msdk/
          thrid/okhttp/Call;Lcom/mbridge/msdk/thrid/okhttp/EventListener;)V
    iget-object v0, p0, Lcom/mbridge/msdk/thrid/okhttp/internal/connection/
          RealConnection;->route:Lcom/mbridge/msdk/thrid/okhttp/Route;
    invoke-virtual {v0}, Lcom/mbridge/msdk/thrid/okhttp/Route;->socketAddress()
          Ljava/net/InetSocketAddress;
    move-result-object v0
    iget-object v1, p0, Lcom/mbridge/msdk/thrid/okhttp/internal/connection/
          RealConnection;->route:Lcom/mbridge/msdk/thrid/okhttp/Route;
    invoke-virtual {v1}, Lcom/mbridge/msdk/thrid/okhttp/Route;->proxy()Ljava/net/
          Proxy;
    move-result-object v1
    iget-object v2, p0, Lcom/mbridge/msdk/thrid/okhttp/internal/connection/
          RealConnection;->protocol:Lcom/mbridge/msdk/thrid/okhttp/Protocol;
    invoke-virtual {p7, p6, v0, v1, v2}, Lcom/mbridge/msdk/thrid/okhttp/
          EventListener;->connectEnd(Lcom/mbridge/msdk/thrid/okhttp/Call;Ljava/
          net/InetSocketAddress;Ljava/net/Proxy;Lcom/mbridge/msdk/thrid/okhttp/
          Protocol;)V
    return-void
.end method
```

Figure 15: Mintegral SDK using OkHttp Platform class's isClearTextTrafficPermitted function, with a triggerable exception which can allow clear-text traffic from the app.

```
private static boolean isValidLink(java.security.cert.X509Certificate p2, java.
      security.cert.X509Certificate p3) {
        if (p2.getSubjectX500Principal().equals(p3.getIssuerX500Principal())) {
            try {
                p3.verify(p2.getPublicKey());
                return true;
            } catch (java.security.GeneralSecurityException e) {
                return false;
            }
        } else {
            return false;
        }
    }
```

Figure 16: isValidLink() function using JSSE's verify to check if the certificate is correctly signed by its issuer. In the case where the verify() function in unreachable, the certificate is considered invalid.

```
public void start() {
    super.start();
    this.setEnabledCiphers(this.enabledCiphers);
    int originalTimeout = this.socket.getSoTimeout();
    this.socket.setSoTimeout(this.handshakeTimeoutSecs * 1000);

    try {
        SSLParameters sslParameters = new SSLParameters();
        sslParameters.setEndpointIdentificationAlgorithm("HTTPS");
        ((SSLSocket) this.socket).setSSLParameters(sslParameters);
    } catch (NoClassDefFoundError | NoSuchMethodError e) {
        if (this.httpsHostnameVerificationEnabled) {
            SSLParameters sslParameters = new SSLParameters();
            sslParameters.setEndpointIdentificationAlgorithm("HTTPS");
            ((SSLSocket) this.socket).setSSLParameters(sslParameters);
        }
    }

    ((SSLSocket) this.socket).startHandshake();

    if (this.hostnameVerifier != null && !this.httpsHostnameVerificationEnabled)
          {
        SSLSession session = ((SSLSocket) this.socket).getSession();
        if (!this.hostnameVerifier.verify(this.host, session)) {
            session.invalidate();
            this.socket.close();
            throw new SSLPeerUnverifiedException(
                "Host: " + this.host + ", Peer Host: " + session.getPeerHost()
            );
        }
    }

    this.socket.setSoTimeout(originalTimeout);
}
```

Figure 17: start() function using setSSLParameters and setEndpointIdentificationAlgorithm to enforce secure hostname verification. This includes a simple check to see if the JSSE's setSSLParameters and setEndpointIdentificationAlgorithm are reachable, and if not implement an alternate path to ensure secure connection.

```
private boolean isOCSP(java.security.cert.TrustAnchor p4, java.security.cert.
      CertPath p5)
    {
        try {
            java.security.cert.CertPathValidator v0_1 = new java.util.HashSet();
        } catch (java.security.NoSuchAlgorithmException) {
            return 0;
        }
        v0_1.add(p4);
        java.security.cert.PKIXParameters v4_3 = new java.security.cert.
              PKIXParameters(v0_1);
        java.security.cert.CertPathValidator v0_4 = java.security.cert.
              CertPathValidator.getInstance(runtime.Strings.StringIndexer.
              _getString(11460));
        int v1_1 = ((java.security.cert.PKIXRevocationChecker) v0_4.
              getRevocationChecker());
        v1_1.setOptions(java.util.EnumSet.of(java.security.cert.
              PKIXRevocationChecker$Option.ONLY_END_ENTITY));
        v4_3.addCertPathChecker(v1_1);
        v4_3.setRevocationEnabled(1);
        v0_4.validate(p5, v4_3);
        return 1;
    }
```

Figure 18: isOCSP() function using JSSE's getRevocationChecker to check the certificate revocation status, if the method is not reachable the OCSP status returns as false.