An In-Depth Analysis of Android's Java Class Library: its Evolution and Security Impact

Timothée Riom, Alexandre Bartel
Department of Computing Science
Umeå University
Umeå, Sweden
{timothee.riom, alexandre.bartel}@cs.umu.se

***Abstract*—Android is an operating system widely deployed especially on devices such as smartphones. In this paper, we study the evolution of OpenJDK Java Class Library (JCL) versions used as the basis of the Dalvik Virtual Machine (DVM) and the Android Runtime (ART). We also identify vulnerabilities impacting OpenJDK JCL versions and analyze their impact on Android. Our results indicate that the complexity of the Android JCL code imported from OpenJDK increases because: (1) there is an increase in the number of classes imported from OpenJDK, (2) there is an increase in the fragmentation of the JCL code in Android as code is increasingly imported from multiple OpenJDK versions at the same time, and (3) there is an increase in the distance between the JCL code in Android and OpenJDK as, for instance, Android developer introduce customizations to the imported code. We also observe that most OpenJDK vulnerabilities (80%) are not impacting Android because the vulnerable classes are not imported in Android. Nevertheless, Android does import vulnerable code and little is done to patch this vulnerable code which is only "patched" when a newer version of the vulnerable code is imported. This means that the code can stay vulnerable in Android for years. Most of the vulnerabilities impacting Android (77%) have a security impact on the availability of the system. By developing a proof-of-concept, we show that OpenJDK vulnerabilities imported in Android do have a security impact. We suggest to seriously take into account public information available about OpenJDK vulnerabilities to increase the security of the Android development pipeline.**

***Index Terms*—Android, external dependency, vulnerability management, managing code complexity, Java, OpenJDK, Similarity Analysis, Vulnerabilities, Security**

I. Introduction

The Android operating system is an open source project which has been scrutinized by researchers on many aspects such as its permission system [1], [2], [3], [4] or the API interface between Android and Android applications [5], [6], [7], [8]. Android applications have access to the application API which is in short a combination of Android-specific Java code and to the runtime of the Android operating system whose upper layers are implemented in Java. This Java code is compiled to Dalvik bytecode to be executed by either the Dalvik virtual machine (up to Android 4.4 in 2013), or Android Runtime (ART)[1] which is the default VM since 2013. While everything at runtime is Dalvik, part of the actual code that is used by the virtual machine is imported from external Java repositories such as OpenJDK, a free and open-source implementation of the Java Platform. While it is well documented that Google has had to handle legal issues against Oracle regarding licencing the Java API [9], not much research has been done on understanding how the Java code surrounding the virtual machine, called Java Class Library (JCL), or Core Libraries (a.k.a. *libcore*) in Android, is actually managed in Android. In this paper, we investigate Android's external dependencies on external JCL repositories and highlight how this code is managed in Android, how it has evolved, and how upstream Java security vulnerabilities impact the Android system.

One of the first objectives of the Android Platfom was to *offer immediate compatibility to the standard Java ME"* [10]. Because of disagreements with Sun, the company behind Java which is owned by Oracle since 2010, Google decided not to use Sun's implementation of the JVM [11]. Instead, Google developed a new virtual machine called Dalvik and use parts – such as files from the OpenJDK – of an open-source implementation of the Java Virtual Machine called Harmony, managed by the Apache foundation. The Apache foundation stopped maintaining Harmony in 2011 because the main participant, IBM, decided to shift its efforts to Oracle's OpenJDK [12]. Therefore, Google maintained the parts of Harmony they use in Android for the next 5 years. In 2016, Android reached version 7.0 (codename Nougat) and the Harmony code it relied on was replaced by OpenJDK's. In 2021, efforts started to replace OpenJDK 7 with OpenJDK 11 in Android 13. This simplified view on the evolution of external dependencies used for the JCL code in Android are illustrated in Figure 1.

Looking at the code of Android 13 we observed that there is no simple mapping from a given JCL Java class used in Android to its corresponding OpenJDK version. In fact, our analysis reveals that Android 13's libcore contains Java classes coming from multiple different OpenJDK versions at the same time on top of containing Java classes that have been customized by Google engineers and which are thus unique to Android. This situation makes it challenging to keep track of which Java version is used where in the Android code base and also makes it challenging to maintain the Java code in Android. Therefore, securing the Android Open Source Project's Java Class Library might not be as trivial as updating the JCL that Android partially imports from external projects. This

[1]https://source.android.com/docs/core/runtime

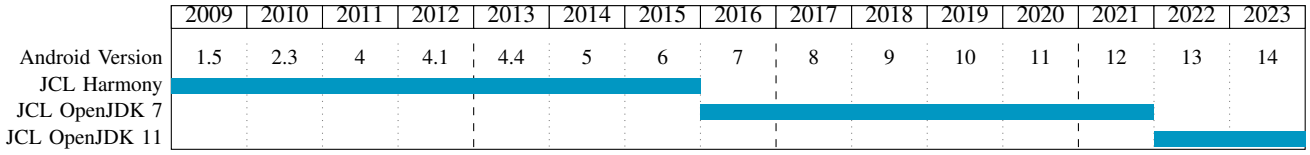


Fig. 1: Simplified view of the evolution of the external Java dependencies used by Android, by year and Android version.

could potentially result in having vulnerabilities that are not patched quickly, or even not patched at all. This translates to extended exposure windows in comparison to OpenJDK and unnoticed vulnerabilities. Not all vulnerabilities impacting OpenJDK have a security impact on Android. For instance, Android does not rely on OpenJDK's sandbox mechanisms, so vulnerabilities to escape the sandbox cannot be used to escape the Java sandbox in Android because it is not implemented nor used in Android. On the other hand, if a vulnerability affecting a cryptographic component of OpenJDK is also used in Android, it might affect Android applications relying on this component.

In this paper, we study the evolution of the JCL in Android and answer to the following questions:

> ***RQ 1.*** *Which OpenJDK versions are used in Android's libcore? How much do they diverge from the OpenJDK upstream?* Our results indicate that the complexity of the Android JCL code imported from OpenJDK increases because there is an increase: (1) in the number of classes imported from OpenJDK; (2) in the fragmentation of the JCL code in Android; and (3) in the distance between the JCL code in Android and OpenJDK.
>
> ***RQ 2.*** *How have OpenJDK vulnerabilities been managed in libcore?* We observe that most OpenJDK vulnerabilities (80%) do not impact Android. The imported vulnerabilities are only "patched" when a newer version of the vulnerable code is imported. The vulnerable code can stay present in Android for years.
>
> ***RQ 3.*** *What is the security impact of OpenJDK CVEs affecting Android?* We observe that most of the vulnerabilities impacting Android (77%) can have a security impact on the availability of the system. We develop a proof-of-concept showing that vulnerabilities can be reached on Android through the Java API.

To the best of our knowledge, it is the first time that the impact of OpenJDK's security on Android is studied. The full content of data, code and results can be found at https://github.com/software-engineering-and-security/AndroidsJCL-SecDev23. The remaining of the paper is organised as follows. The reader familiar with Java or Dalvik virtual machines can skip the background Section II. In Section III, we present the threat model considered in this paper. In Section IV, we present our methodology. Section V presents the results and answers to the research questions. We discuss the limitations of our approach in Section VI. We present the related work in Section VII. Finally, we conclude and present the future work in Section VIII.

II. Background

In this section, we introduce the concepts used around the Java environment as well as the terminology used around the Dalvik environment used in Android and explain how elements of the Java environment are connected to the Dalvik environment.

A. Java Code Execution

A Java Virtual Machine (JVM) executes Java programs whose code is represented by bytecode instructions. The Java Runtime Environment (JRE) contains the JVM and the Java Class Library (JCL). The JVM is often represented as the `java` binary under Unix systems. The JCL is a set of core Java classes that are shipped with the JVM and on which any Java developer can rely when writing code. Typical examples of JCL classes are `java.lang.Object` and `java.lang.String`. JCL classes are automatically loaded when the JVM starts and are available to the Java program. There exist several implementations of the JVM, the most famous being Oracle's OpenJDK.

B. Android Code Execution

Android applications are written in Java and compiled into Dalvik bytecode, a representation very close to the Java bytecode. There are some differences such as the use of registers instead of the stack for computations [13]. Dalvik bytecode was executed by the Dalvik Virtual Machine (DVM) until Android 5.0. After Android 5.0, the bytecode is executed by the Android Runtime (ART). In all cases, whether executed with the DVM or ART, the application code generated from Java needs to access JCL classes. In Android, JCL classes are imported from external dependencies such as OpenJDK. In Android, the Core Java classes, which include the imported JCL classes, are grouped in a component called *libcore*. In this paper, we focus on studying the part of Android's libcore that is imported from external JCLs.

III. Threat Model

The JVM has – actually *had* because it has been removed since Java version 17 [14] in 2021 – a sandbox mechanisms called the "Java sandbox" which could be used to limit the permissions of untrusted code. For instance, the untrusted code could run without permission to access the filesystem. However, many flaws have been identified in this sandbox mechanisms which made it useless in most of the cases

because an attacker could bypass the restrictions [15], [16]. Therefore, Android's security architecture designers decided not to rely on these mechanisms to restrict the code running in Android applications. Instead, Android relies on, among other things, the security mechanisms provided by the Linux operating system such as user ID and groups to restrict what an application can do (ex: only applications whose user ID is in the group CAMERA can use the camera of the device). Hence the question: do vulnerabilities discovered in the JVM impact the Android OS and/or Android applications? The answer is that it depends on the vulnerability.

If the vulnerability is targeted at escaping the Java sandbox, which Android does not use, then probably that CVE is of no impact to the security of Android and its applications. Such attacks also suppose that the target VM executes untrusted code downloaded from a remote host on the Internet, which is, to the best of our knowledge, not common for most of the Android applications. Therefore, we do not consider this attack vector as important in this paper. On the other hand, if the vulnerability can be triggered through a Java API, then it could also impact Android applications that have untrusted input reaching the vulnerable Java API. A typical example is a denial of service (DoS) attack for which the attacker generates a specific payload to send to the vulnerable API to hang the targeted system. If the JVM is vulnerable to such an attack, and if all the vulnerable code is also in Android, then Android applications relying on this vulnerable code could be impacted. In this particular example with a DoS vulnerability, the Android application would hang. With a more critical vulnerability such as remote code execution (RCE), an attacker could execute arbitrary code as the vulnerable application and potentially attack other components of the Android system.

IV. Methodology

This methodology section describes our approach to answer the different research questions. Note that in the research we conduct in this paper, we focus on the analysis of OpenJDK and do not consider Apache Harmony, which has only been used in Android versions before 2015.

A. ***RQ1:*** *Which OpenJDK versions are used in Android's libcore? How much do they diverge from the OpenJDK upstream?*

For each Java class JC_{AR_i} in libcore for a given Android version AR_i released at date D, we identify JC_{OP_j} in every OpenJDK version OP_j using the Java class Fully Qualified Name (FQN). We consecutively compute the distance with JC_{OP_j} released before D.

In other words, for each Android Java class that is also present in OpenJDK, we compute the distances between versions of the same class across OpenJDK's history[2] and identify the closest. The signatures of each Java class of libcore, and of each class of OpenJDK are generated using `tlsh_unittest`. This **L**ocally **S**ensitive **H**ash generator [17] computes a digest with a sliding window of 5 bytes. It is composed of a 3-byte header and a body. For the body, each 5 bytes window is used to generate buckets of 3 bytes, and the population of buckets gives quartiles. Each bucket, depending on its position w.r.t. the quartiles, provides a value (`00`,`01`,`10`,`11`) that, added one after the other, produces the digest body. A distance can be computed from a digest to another using a derivative from the Hamming distance. TLSH has been adopted by Virus Total[3] and Malware Bazaar[4] as a similarity technique. TLSH is also part of Structure Threat Information eXpression (STIX) 2.1[5]. Our comparison provides, for each Java class in Android's libcore, a closest OpenJDK origin. It allows us to generate a profile for each Android version and observe the evolution of the OpenJDK versions used in Android.

While there is a closest version JC_{OP_j} for each JC_{AR_i}, the distances are different depending on JC_{AR_i} and on AR_i. Therefore, we also profile the distances over each version AR_i. In other words, we perform an analysis on the evolution of distances between classes in Android and OpenJDK for all Android releases.

B. ***RQ2:*** *How OpenJDK vulnerabilities have been managed in libcore?*

To be able to answer this question, we need to collect vulnerabilities (CVEs) affecting OpenJDK and then analyse libcore's code to understand if any of these vulnerabilities are also present in Android and for how long. With this information, we can compute the window of exposure for each vulnerability, i.e., the time during which vulnerable code is present in the Android system. We particularly seek vulnerabilities that are left unpatched in Android while patched in OpenJDK, i.e., overexposing Android.

As illustrated in Figure 2, the procedure to identify Java CVEs present in Android features five steps. In the first step (**CVE collection**), we gather the list of CVEs from the NVD NIST website (through the JSON feeds [6]) that affected any of the aforementioned OpenJDK versions. Some CVEs affect several OpenJDK versions; hence, we keep track of this information to later compare patched files with OpenJDK's version of the same file. We use the *cpe*[7] field for either the string `oracle:openjdk`, or `sun:openjdk`. We then collect and sort CVEs and affected platforms. In the second step (**Patch collection**), as there is not one single source of information capable of providing all the patches, we adopt different strategies.

- We first inspect the Bugzilla web-page related to this CVE, if it is available, and provides a link to the patch next to the characteristic *Upstream Commit* expression.
- Or we try to collect the *Bug Id* in the description of the Mitre or NVD's webpage. We then query the Mercurial

[2]https://www.java.com/releases/fullmatrix/

[3]https://developers.virustotal.com/reference/files-tlsh

[4]https://bazaar.abuse.ch/api/#tlsh

[5]https://docs.oasis-open.org/cti/stix/v2.1/os/stix-v2.1-os.html

[6]https://nvd.nist.gov/vuln/data-feeds

[7](Common Platform Enumeration)

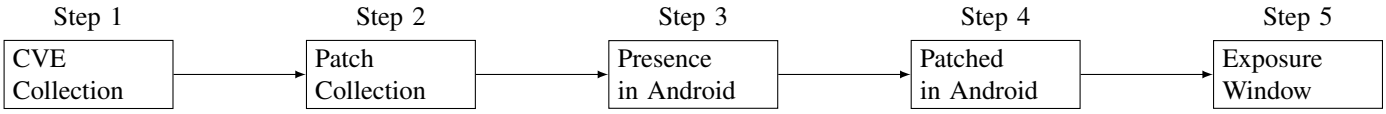


Fig. 2: Steps to identify Java CVEs present in Android and their Lifetime

repository of OpenJDK to collect the hypertext link to the commit.
- Or we query through a web search engine until we can retrieve a *Bug Id*. This last step can require the help of the WayBack Machine for old OpenJDK CVEs.

A CVE might impact Android if the classes affected by the patch for the CVE in OpenJDK are present in libcore. Therefore, in Step 3 (**File presence in libcore**), we identify CVEs for which all patched files are also in Android's libcore. Note that we only consider filenames at this point, and we do not yet know if the Java files are patched in Android. From the commit web page, we collect the list of files modified by the commit. We retrieve them through the Git history of libcore, following renaming, up to the introduction of OpenJDK 7 in libcore. If, for a CVE, not all files match, we discard the CVE since it cannot impact Android. Once we have CVEs for which all files are present in libcore, we manually check if there is any trace of the OpenJDK patch in all the concerned Android releases. This fourth step (**Find patched classes in libcore**) identifies from which Android version an OpenJDK CVE is patched in libcore and, if so, when. Finally, in Step 5 (**Exposure window overhead computation**), we compare the date of the patch in OpenJDK with the date of the patch in AOSP/libcore to compute the overhead window exposure for each selected CVEs.

C. ***RQ3:*** *What is the security impact of OpenJDK CVEs affecting Android?*

We first analyse the set of OpenJDK CVEs impacting Android and characterise them in terms of attack vector and impact of security. Then, we report on our discussions with the Android security team to understand why these vulnerabilities have not been patched. Finally, we present a proof-of-concept to exploit a real OpenJDK vulnerability affecting Android to show that they actually impact the security of the Android system.

V. Results

A. ***RQ1:*** *Which OpenJDK versions are used in Android's libcore ? How much do they diverge from the OpenJDK upstream ?*

We first look at the origin of Java classes in Android and determine from which versions of OpenJDK they come from in Section V-A1. Then, in Section V-A2, we observe how close the Java classes in Android are to their equivalent classes in OpenJDK.

1) Mapping between Android Java classes and OpenJDK's:

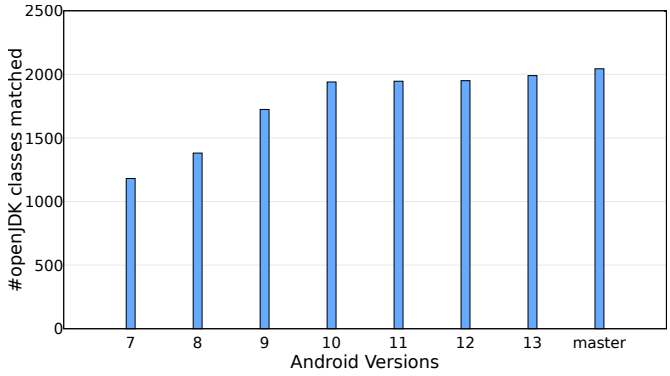
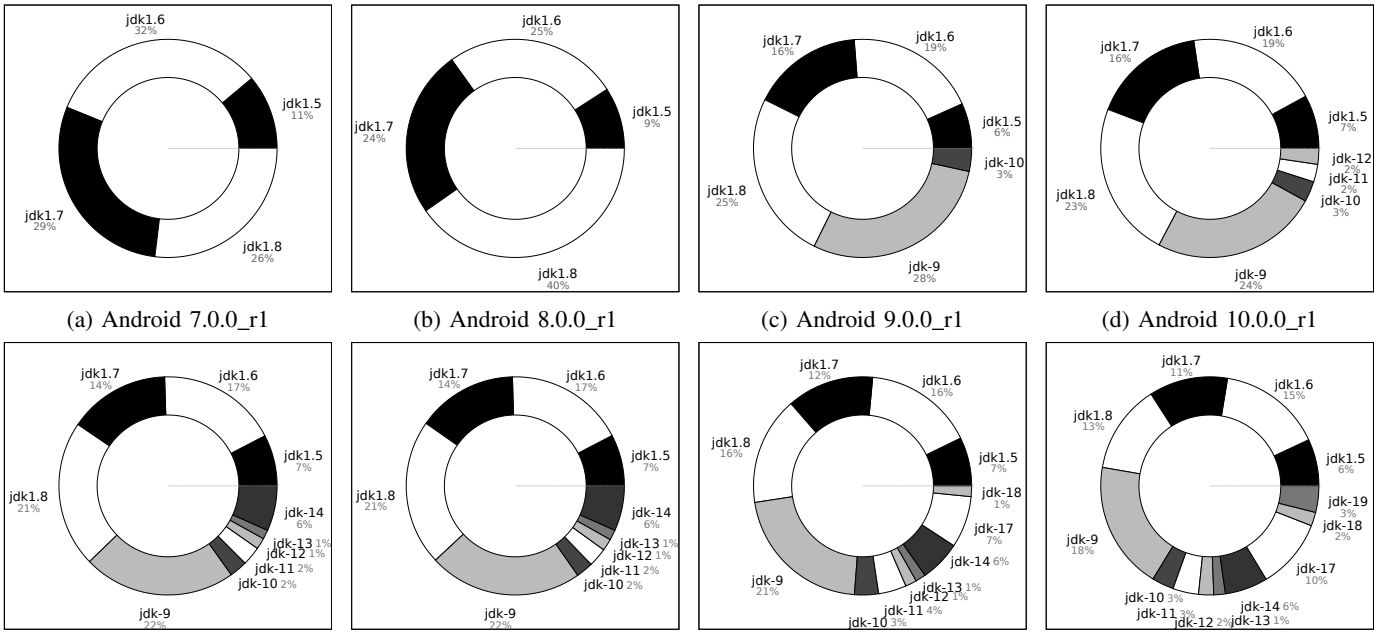


Fig. 3: Number of matched OpenJDK classes per Android libcore version

a) Decription: Figure 3 presents the evolution of matchable paths from libcore in available-at-release-time OpenJDK versions. If the number of classes increases at first, from 1.1k to 1.9k, it stabilises from Android 10 onward, to slightly increase in late versions and the master branch, reaching 2048. Hence, almost doubling since Android 7.

We present in Figure 4, the results of the closest version of OpenJDK for every libcore class. We select the OpenJDK versions for which the distance from digest to digest (as computed by `tlsh`) is the smallest. Among those OpenJDK versions with the same distance to the version in libcore , we select the earliest. We see that Android 7's and 8's libcores provide classes closer to OpenJDK 1.5, 1.6, 1.7 and 1.8. The proportions of classes in 1.7 and 1.8 increases from one version to the other, passing from 56,1% to 67,8%. The most noticeable increase concerns jdk1.8, passing from 25,5% to 42,2%. In Android 9, about one third (32,2%) of the found JDK classes are closer to the newly available JDK versions: jdk-9 and jdk-10. The proportion of classes closer to jdk1.5 and jdk1.6 decreases from 44% in Android-7, and 36% in Android 8 to 25% of the Java classes. The trend is overall similar in Android 9's libcore and in Android 10: newly available versions jdk-11 (Long Time Support, LTS, version) and jdk-12 are the closest versions for only a low proportion of Java classes. Overall, Android 12 has a similar profile to Android 11. In both cases, jdk-14 takes a more significant proportion than previously introduced OpenJDK versions (taken independently). Since Android 13, Java classes based on latest versions of OpenJDK are more common, underlining a rejuvenation of Android's OpenJDK based classes. Versions succeeding jdk-10 account for around one third (36%) of the Java classes. OpenJDK-17 is the most represented one among



(a) Android 7.0.0_r1 (b) Android 8.0.0_r1 (c) Android 9.0.0_r1 (d) Android 10.0.0_r1

(e) Android 11.0.0_r1 (f) Android 12.0.0_r1 (g) Android 13.0.0_r1 (h) master branch

Fig. 4: Representation of the OpenJDK profiles of Android's libcore over versions

those with 10%.

b) Investigation: In late Android versions, very old OpenJDK versions still appear, as jdk1.5 and jdk1.6. jdk1.5 represents almost always the same proportion of classes, regardless of the absolute evolution of Java classes in libcore. This partially explains through the selection of the closest in distance and earliest in release date, version of OpenJDK, in order to represent Figure 4. Hence, if a Java class did not change over time, then the distance is the same when comparing the libcore class. Among the 197 classes that are ever closer to jdk1.5 across Android versions, 84 are always closer to the 1.5 version of OpenJDK. Among the most concerned directories, 13 classes are always closer to jdk1.5 in `w3c/dom` and 7 classes under `java/lang`. For jdk1.6, 521 classes are at some point computed closer to jdk1.6 and 198 classes are always closer to jdk1.6. Among the directory, 25 classes are always under jdk1.6 in `java/sql`, 17 in `java/security` and 13 under `security/cert`. For instance, `java/security/BasicPermission.java` is the same for all jdk1.8 versions. `java/nio/ch/Groupable.java`'s digest has the same distance with every OpenJDK versions since its introduction in jdk-9. Hence, we select jdk-9 while the latest version the class was imported from could be jdk-11. Last example, `java/net/CookiePolicy`, is equidistant from every version of OpenJDK since OpenJDK-1.8.0_05.

2) Distances between Android and OpenJDK classes: In this subsection, we aim to analyse the evolution of the distances between Java files in libcore from the version they are the closest to. Figure 5 provides, for each Android version, the proportion of Java classes that have a distance below 10,

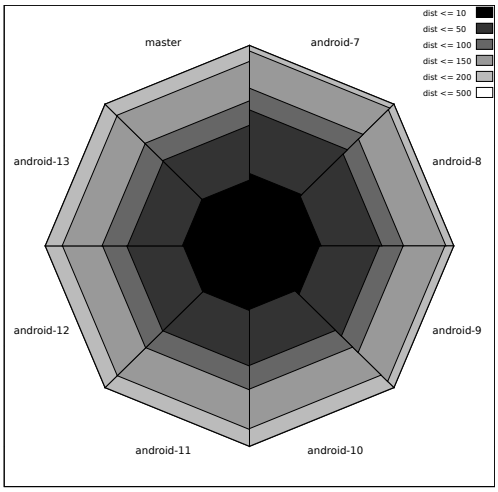


Fig. 5: Divergence of libcore Java classes from OpenJDK closest versions

50, 100, 150, 200, and 500 from the closest JDK version of the same class. We provide a few examples and definitions for the reader to grasp the concept of a 'distance' in our context:

- "*a distance score of 0 represents that the files are identical (or nearly identical)*" [17]
- In Android 13, `java/math/MutableBigInteger.java` has a distance with jdk-17 of only 1. It corresponds with 1 small `if` block (passing from 1 to 6 lines) and 3 altered comment lines over a file of 2.3k lines.
- In Android master version, the file `java/sun/security/x509/AVA.java` has a distance of 10

with its counterpart in jdk-11.0.6. It corresponds to 15 lines changed, 1 section of 11 lines added in Android, 4 shorts blocks of code changed specifically for Android, and one block of 8 lines commented in Android. This class contains 1.4k lines.
- Still in master, the Java file `java/sun/nio/fs/AbstractPoller.java` has a distance of 50 with jdk-19. Considerably less alteration can be observed, with 7 lines altered, yet in a file containing 290 lines only.

The darker the Android release related section, the shortest the distance is between the represented libcore classes and classes in OpenJDK. For instance, we observe that Android 7 is the version closest to OpenJDK ;it has the most classes with the shortest distances among all Android versions. This observation can be explained by the fact that Android 7 is the version introducing OpenJDK in Android, thus containing few modifications by Android developers. We further observe that the distances increase from Android 8 until Android 10, for which the least Java classes are closer to the closest OpenJDK version. This also corresponds with the introduction of new Java classes, as seen in Figure 3. Starting at Android-10 and up to the latest Android version, we observe that the distances do not change significantly.

The Java classes with the highest distance from their closest OpenJDK version are always the same across the versions of Android. Among these, we find, for instance, `java/lang/invoke/MethodHandle.java` or `java/net/URI.java`. More surprisingly, in Android 10 `java/lang/String.java` is part of this group of classes for which the closest version is at a distance above 500. This might be explained by the fact that this class is highly used by Java programs and that Android developers have made significant modifications to optimize its code.

> By analysing the evolution of the OpenJDK Java Library Class in Android's libcore since its introduction, we observe that (1) there is an increase in the number of Java classes imported from OpenJDK, (2) there is an increase in the distances between libcore classes and OpenJDK's, and (3) there is an increase in the fragmentation of OpenJDK versions used in Android. These observations highlight an increase of the complexity of Android's libcore JCL. This complexity increases the difficulty to maintain the JCL code in Android and might also introduce unintended side-effects such as unforeseen security issues.

*B. **RQ2:** How OpenJDK vulnerabilities have been managed in libcore?*

To help following up the CVE selection process, we have summarised figures in Table I. From the NVD feeds, we extract 82 CVEs affecting OpenJDK[8]. Nine of these CVEs impact OpenJDK17, the latest version, while 73 impact older versions.

TABLE I: Summary of the identification of OpenJDK CVEs concerning files present in Android

CVEs affecting OpenJDK	82	100%
CVEs for which Android contains at least 1 of the patched files	25	30,5%
CVEs for which Android contains all patched files (at least once in a version OpenJDK)	18	22%
CVEs for which Android contains all patched files (in all Android versions with OpenJDK)	16	19,5%
CVEs presenting an exposure overhead in Android's history	13	15,9%

Patch Collection Bugzilla provides an issue tracker page for 80 of these vulnerabilities and provides link to the patch for 54 of them. Recall that we need information about the patch, since we assume that an OpenJDK vulnerability is present in Android if all the Java files affected by the patch are present in Android. We retrieve 7 more fixing-commit references through the *Bug Id* contained in the descriptions. It leaves 21 vulnerabilities for which a patch could not be found. We manually find the commit for 5 of these CVEs using the WayBack Machine[9], following broken links, to find the bug id and then the commit hash. 11 CVEs affect a library that we did not find in libcore (*littleCMS*, *libxml2*, *pulseaudio*, *gstreamer*, ...). Finally, one CVE is undisclosed, and we cannot reach the fixing commit for 4 CVEs. In total, we have identified how 66 CVEs are patched in OpenJDK.

File presence in libcore From the 66 CVEs for which the fixing commit hash could be found, there are 25 CVEs for which at leas one file impacted by the fixing commit is found in libcore and 18 CVEs for which all the files impacted by the fixing commit are present in libcore. Considering that Android needs to present all files to risk exposure, we continue with this list of 18 CVEs. For 2 CVEs (CVE-2020-2781 and CVE-2021-35550), the files (`sun/security/ssl`) were present in libcore for Android 7 but have been removed from it in the following main releases. Thus, these vulnerabilities do not lead to an over-exposure and we omit them in further steps.

Find patched classes in libcore In total, 16 CVEs for which all files that need a patch in OpenJDK are present in at least one version of libcore. These CVEs are from 2020 and later, hence relatively recent w.r.t. the span of Android major releases considered (from 2016) and of OpenJDK considered vulnerabilities. We manually verify when the patch was from and if the classes in libcore seem patched.

One CVE, namely CVE-2020-2593, presents a particularity: the area patched in 2019 in the upstream had been altered in 2017 for libcore, before it was considered vulnerable in OpenJDK. In detail, with Android 9, the method `isBuiltinStreamHandler` appears in `java/net/URL.java`. However, Android developers have introduced specific modifications to this file so to better use Android specific protocols. These modifications have "removed" the vulnerable parts in the code. Therefore, the exposure window overhead is nil for this CVE, as seen on Figure 6.

[8]last retreived on March 24th 2023

[9]https://web.archive.org/

Regarding CVE-2022-21341, the file is closer to OpenJDK1.8, and the patch correcting it in OpenJDK dates from September 27th 2021. The code of `java/io/ObjectInputStream.java` reveals however that from Android 9 onward, a serialisation issue was met, monitored by a specific unitary test ((SerializationStressTest#test_2_writeReplace)) and addressed already back in 2018 at the exact location that needs patching. Hence, the vulnerable code portion was already modified before being introduced in Android. Therefore, there is no exposure of the vulnerability in Android. The code that needs to be patched for CVE-2022-21283 is never present in libcore before Android 13, and is straight modified by the Android development team for integration in the architecture. Hence, Android seemingly never contained the portion of the code vulnerable in OpenJDK, and cannot have been over-exposed to it.

Exposure window overhead

Figure 6 represents the exposure overhead of known OpenJDK CVEs that are present in libcore. We represent for these 16 CVEs the duration from the year of the patch in OpenJDK to the date of the newest Android release that do not present the vulnerable code anymore. We observe that it has taken at least 2 years to remove known CVEs from libcore and that 4 of them are not patched in any current release version of Android but are removed in the master-branch. We also note that 8 CVEs still are present as unpatched in the master branch as of May 2023.

To understand if CVEs have been patched or only "removed" by using a newer version of the impacted Java files, we look at AOSP's master Git branch and trace back the changes of files of concern across Android versions up to Android 7, the first with OpenJDK. We do not observe commits specifically removing a CVE, nor commits close to commits patching CVEs in OpenJDK. On the other hand, we observe that the Java files impacted by a CVE are updated to newer versions, mostly for stables releases of OpenJDK 11 (CVE-2021-35561) or 17 (CVE-2020-2803, CVE-2020-2830 and CVE-2022-21340).

> Most of the vulnerabilities affecting OpenJDK are not present in Android because the code is not never fully imported (80.5%). Nevertheless, it seems that OpenJDK vulnerabilities are not taken into account in Android in general and that they are never intentionally patched. They "disappear" when a newer OpenJDK version, in which they are patched, is deployed on the Android system.

C. ***RQ3:*** *What is the security impact of OpenJDK CVEs affecting Android?*

Table II lists the 16 CVEs from OpenJDK that we have identified as being present in at least one Android version. We observe that 3 CVEs (CVE-2020-2803, CVE-2020-2805 and CVE-2021-2341) can only be triggered with untrusted code running in the Java sandbox. First, we assume that it is not

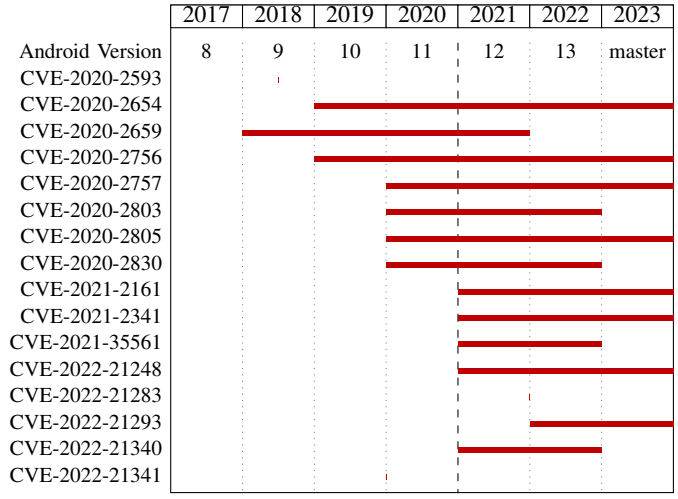


Fig. 6: Lifecycle of OpenJDK CVEs present in Android. In total 13 CVEs impact at least one Android version. 12 CVEs are still present in the latest Android release 13. 8 CVEs are still present in the "master" Git branch of AOSP as of May 2023.

TABLE II: List of OpenJDK CVEs impacting Android. For each CVE is listed the impact score (between 0 and 10, 10 being the highest security impact), the security impact (C: confidentiality, I: integrity, A: availability), if the vulnerability can be triggered from code running in the Java sandbox (Y/N), if the vulnerability can be triggered from an API (Y/N), if it requires human intervention (Y/N) and if it impacts the latest Android release (Y/N). The rows with a gray background represent CVEs which can be triggered through the Java API in Android.

CVE	Score /10	Security Impact	Sandbox	API
CVE-2020-2593	4.8	CI	Y	Y
CVE-2020-2654	3.7	A	Y	Y
CVE-2020-2659	3.7	A	Y	Y
CVE-2020-2756	3.7	A	Y	Y
CVE-2020-2757	3.7	A	Y	Y
CVE-2020-2803	8.3	CIA	Y	N
CVE-2020-2805	8.3	CIA	Y	N
CVE-2020-2830	5.3	A	Y	Y
CVE-2021-2161	5.9	I	Y	Y
CVE-2021-2341	3.1	C	Y	N
CVE-2021-35561	5.3	A	Y	Y
CVE-2022-21248	3.7	I	Y	Y
CVE-2022-21283	5.3	A	Y	Y
CVE-2022-21293	5.3	A	Y	Y
CVE-2022-21340	5.3	A	Y	Y
CVE-2022-21341	5.3	A	Y	Y

realistic for an attacker to exploit them in Android since the Java sandbox is not used. Second, we do not consider them since they do not fit within our threat model (see Section III).

For the remaining 13 CVEs, 10 (77%) have an impact on availability, 3 (27%) on integrity, and 1 (7%) on confidentiality. This observation means that for most vulnerabilities, a potential attacker can only have an impact on the availability of the Android application or system. In practice, this could result, for instance, in an Android component or application not responding to user input.

We reported the list of known OpenJDK vulnerabilities that we have identified to impact Android libcore to the Android security team. To our surprise, these known vulnerabilities were classified by the security team as "not an issue". The reasoning behind this classification is the lack of proof-of-concepts (PoCs) to show that the known vulnerabilities actually have a security impact on the Android system.

Reverse engineering CVEs to produce PoCs can be very challenging and time consuming. Nevertheless, we decided to reverse engineer one CVE to show that our assumption that "if all Java files impacted by a CVE in OpenJDK are present in Android then the vulnerability can be exploited in Android" holds. We successfully developed an exploit for CVE-2022-21340 and packaged the code in an Android application. Executing the Android application with the untrusted input triggering the vulnerability through the Java API results in the Android application being unresponsive. Thus, it clearly shows that the vulnerability does have a security impact in Android; in this case an impact on the availability of the Android application. This confirms our hypothesis and shows that known OpenJDK CVEs can impact the security of the Android system and that they should be considered in the Android development pipeline. The PoC has been sent to the Android security team and is currently under analysis.

A majority of OpenJDK vulnerabilities affecting Android (77%) can have a security impact on the availability of Android OS components or applications. We have demonstrated that this observation is not only theoretical and that OpenJDK vulnerabilities can have a real security impact on the Android system by developing a proof-of-concept exploiting an OpenJDK vulnerability in Android. Known OpenJDK vulnerabilities should be taken into account in the development pipeline of Android and patched as soon as possible.

VI. DISCUSSION

A. Delay for Integration of newest Java versions in libcore

When libcore changed from Apache Harmony to OpenJDK, in 2016, version 8 LTS (Long Term Support) had already been released for over 2 years, and Android 7, the latest LTS at that stage, for 5 years. Unsurprisingly, these versions are selected as the closest version for more than half of the classes in both Android 7 and 8. We can however note that certain classes and features of Java 8 are not supported by Android as, for the APIs available to Android Applications, Google needed to desugar[10] those. Hence, only by 2017[11] these features were properly available in all Android API, by switching `jack` for `javac`.

Android started to adopt the next LTS version, OpenJDK-11, with even more delay. While it was generally available in 2018, jdk 11 represents only 4% of the classes in the following-year's libcore (Android 10). They represent only 10% in Android 11 and 12, to double in Android 13. Also, only Android 13, 5 years after release, provides a desugared support for Java 11 to applications through APIs[12].

The latest available LTS version, OpenJDK 17, is the closest to already 8% of Android 13's classes, one year after its 2021 release. And in April 2023's master branch, classes closer to version 17 and latests almost double to 15%.

Aside from the struggle to have to re-implement desugared support in order to provide latest support to applications, another aspect might explain the rejuvenation in Android 13 and in the master branch: the end of Active Support for older versions. For Java SE, from which OpenJDK inherits, Java 8's Active Support ended in 2022, and Java 11's Active Support ends in September 2023[13]. Further Java 11's Security Updates aim to end in 2026, with Java 17's Active Support. The before-Android 13 adoption rate of latest versions is not suggesting that OpenJDK 17, and later ones, would have by then been a sufficiently adopted in libcore. Therefore, Android could have been soon without Active Support nor even security updates for Java code its Runtime uses. Android would have then been in a similar position as to when support ended for Apache Harmony.

B. Android Expected Upstream

This will to catch the train of Java releases can be found in the Source Code of Android. In October 2021 and under libcore, an EXPECTED_USPTREAM file appeared. This file lists matching paths for Android 13 onward with OpenJDK [14]. This file creates a straight link from the upstream OpenJDK version from which a file is extracted, and surrounding tools enable its automatic update in Libcore. Hence the development of Android is automating the reduction of the delay between available OpenJDK releases and helps Android be ready for arriving ends of support. For instance, the first java class to follow Openjdk17 was added in February 2022[15]This does not invalid *tlsh* results as manual inspection provides, for instance, a distance of 1 from libcore's `java/math/MutableBigInteger.java` with OpenJDK-17, while the the same file has a distance of 2 with OpenJDK-11, version cited in EXPECTED_UPSTREAM. The modifications between versions 11 and 17 concern 3 `if` and `for` small blocks,

[10]Removal of syntactic language simplifications [18]

[11]https://www.youtube.com/watch?v=LhaSi6_i2bo

[12]https://developer.android.com/studio/write/java11-default-support-table

[13]https://www.oracle.com/java/technologies/java-se-support-roadmap.html

[14]Available only for Android 13 and the master branch, our approach differentiates by matching the paths for all versions of Android, and follows name changes.

[15]`Commit:9d2e868`

below 11 lines each and a few comment lines. Libcore's version only differentiates from the OpenJDK-17 by one 6-line block and 3 comment lines while differentiating from OpenJDK-11 by 2 blocks, including a 11 lines one, and by 2 comment lines. Hence, the libcore version appears, to that regard, closer to the version selected by `tlsh`.

C. Android improvements

Google presents in recent year a noticeable push in reducing vulnerability windows of exposure overhead in the whole Android software stack with Projects Treble [19], Mainline [20], and Android Common Kernels(ACK)[21]. Project Treble, since Android 8, primarily splits the Android OS stack from handset's hardware exploitation code and modularizes Android. Thus, updating Android does not require to recompile it with all the vendors code (hardware and UI). Project Mainline takes advantage of this modularization to implement Over-The-Air (OTA) per-module updates. Modules like the Runtime Module, that gathers the ART and libcore, are updated in the format of an apk-like file (called APEX) through the PlayStore[16]. In those two cases, the window of exposure is reduced from the Android Source code to the download on user's handset. Finally, ACK provides Generic Kernel Images and better aims (as for libcore and OpenJDK) to, both, keep track with releases pace in the upstream and directly benefit from their security patches. Strengthening this link between the upstream library has, for the least, two security benefits. It first enables to considerably reduce the effort to correct vulnerabilities: the source code of the project can directly inherit from the issue correction from the library itself. The automation of the update also reduces the risk of overexposing the code to a vulnerability publicly released in the library.

D. Threats to validity

TLSH tool: Our analysis relies on the results provided by the `tlsh` tool. The tool provides a combination of advantages that enabled its broad adoption for similarity analyses. However the computation of a digest makes complex the explanation of distances comparisons. Furthermore, the documentation [17] indicates that the evolution of the distance by increasing the number of mutations is not a bijection. For example, if a file F is close to two others G and H, the closest file to F might have a slightly higher distance.

Set of vulnerabilities: In Section IV-B, we do not consider a vulnerability if its patch impacts one or more files that we do not find in Android's libcore. Verifying that Android can be exploited through those partial exposure can be addressed in futur work.

VII. Related Work

A. Java

Java is an object-oriented programming language that provides different studied features such as reflection [22], serialisation [23], exception handling [24], or, again, an always improved garbage collector [25]. Java also provides a list of APIs, that tend to break regularly [26], when changes or updates in those APIs infer bugs in codes that call their former form.

The Java Security Model (JSM) is extensively presented in an article that also presents the specific weaknesses Java is more prone to [15]. The security model includes an Access Control architecture that Java developers more often question the community about, better than misusing it [27]. Yet, even with the Security Manager enabled, and claims to provide type safety [28], some vulnerabilities, like CVE-2018-2826, are found, enabling to disable the Security Manager through type confusion. Deserialization, the principle to get back the former state of a Java program after it was saved into a stream of bytes, also allows the execution of gadgets from the Java Class Library through the acceptance of untrusted Data [29]. The aforementioned article [15] exposes Java's weakness profile through 87 publicly available exploits. Most of which enable the "*Unauthorized use of restricted class*", "*Loading of arbitrary classes*" and "*Unauth. definition of privil. classes*". Some involve design weaknesses, requiring proper redesign. Otherwise, single step exploits can still be combined to gain further privileges, and expand their impact on the system. Another aspect of the JSM regards information hiding: preventing the access to some of the system's private variables or methods. Exposed to some extents in [15], a lightweight, measure could reside in isolating these in a black box, using tokens, and would already block 84% of the tested exploits for a 2% overhead [30]. Other, heavyweight measures are discussed, as adding a `critical` status to some fields and methods, or adding runtime and compile time verification routines.

The Runtime Environment executing the Java bytecode is another exposed component. To reveal its bugs and weaknesses, researchers implemented techniques to generate, mutate, and then execute Java bytecode. Yang et al. [31] successfully generated a program from existing test cases for IBM's J9 with bytecode deactivating the Security Manager. Bonnaventure et al. [32] generate Java programs from scratch to trigger type confusion vulnerabilities in the JVM.

B. Android

As mentioned in [10], Android Applications are designed to be developed in Java. In 2017, Google introduced, and since pushes[17], for writing, the applications using Kotlin[18], a programming language co-founded with Jet Brains. This language aims to provide a simpler syntax, yet compatible with the Java bytecode and the JVM, and to be attractive, better-coupling with Android [33], [34], [35]. If it seems to reduce the emergence of *CWE-710: Improper Adherence to Coding Standards*, Kotlin may also induce more *CWE-664: Improper Control of a Resource through its Lifetime* [36]. Android APIs provide the possibility to call and execute,

[16]https://source.android.com/docs/core/ota/modular-system/art

[17]https://developer.android.com/kotlin/first

[18]https://kotlinlang.org/

in applications, lower-level, faster-executed [37] code and features (like C or C++, gdb, and since LLDB with the Clang change[19]) through the Native Development Kit(NDK). Native code of one application that is, to some extents, exposed and reachable by other applications through Direct Inter-app Code Invocation [38].

It is also through Java code that Services and Features are provided to applications in the Android Framework. Toolbox of methods available to the developers to provide user-friendly and resourceful applications. Developers can easily access to the phones camera, bluetooth, run services in the background in a documented way[20]. The use of these APIs is, nonetheless, not a guarantee of code quality. In versions up to Android Honeycomb, it was proven that the use of Android APIs was usually correlated with a higher frequency of defaults fixing commits[39]. Letting the authors wonder if it is because the platform is difficult to use. In another work, the hypothesis is that some APIs are silently modified[40], or changes too often[41].

A review of Android's Security by Google developers was published in 2021 [42]. Up to Android API-level 28 (9.x), they describe the Android security ecosystem. Authors specifically focus on the different access control systems implemented in Android. It is split in three different categories: Android Permissions, Unix Access Control and SELinux Access Control. The higher layers follow a Discretionary Access Policy control based on permissions. Applications reference these permissions (camera access, fine or coarse geo-location) they claim to need in a Manifest.xml file. The system will check, upon request, if the supplicant (e.g., API call, Content Provider or Intent) holds the appropriate permissions. This permissions system has been criticised for its scarce, incomplete or inaccurate documentation [43], [44]. Its hierarchy has been therefore improved and clarified (API-level 17[21]). Permissions can since be granted at runtime, revocable and individualised (API-level 24). The *Unix Access Control* differs from usual User-ID attribution as each process is provided with a different UID, rather than being granted the user's UID and permissions. Since API-level 18 *Security Enhanced Linux* (SELinux) further protects components with a Mandatory Access Control [45], [42]. It was released coincidentally with criticism about the lack of control over the kernel calls [46]. It concerned at first only four processes: `installd`(installation daemon), `netd` (network connectivity daemon), `vold` (volume events daemon) and `zygote` (process creation). Soon it was expanded to all userspace (API-level 21), tightened up (API-level 25) and available for SOCs vendor to write their own further restrictive specific rules(API-level 26). A comprehensive study on software aging across Android versions and vendors demonstrates a constant aging trend of Android Applications [47]. It affects all studied vendors, though more significantly those customising the OS. One particular element correlated with aging is the utilisation by Applications of the *System Server*, and specifically the Garbage Collector.

The switch from Dalvik to ART broke tools [48] enabling applications monitoring, static and dynamic analysis. ARTist [49] overcomes the change from `.dex` to `.oat` files by inserting steps in the optimisation step of the *dex2oat* compiler. Deployed through an application with root access to lure the system into using the modified *dex2oat*, it is then possible to monitor applications through the ART, discriminate third-party libraries behaviours in apps [50], and dynamically analyse Android system servers' system [51]. Regarding Android internals, a fuzzing approach, namely FANS [52], already provides insights on Android Native Services. The tool gathers the AIDL interface declaration and provides inputs that shall violate the documented restrictions over native calls. The dynamic analysis mentioned above [51] uses other fuzzing tools on 2k APIs for over 5minutes each: Chizpurfle [53], AFL [54] and RandFuzz [55]. Their result confirm and enhance previous APIs' permission mappings [4], [56]

The literature agrees that the optimal conditions for the public release of a vulnerability are met when the vendor considers the cost of its clients as its own [57], [58]. It is what Google has been doing, isolating Android from the Original Vendor Manufacturers hardware and middleware, with project Treble [42]. Investing resources to make the Android Stack to update independently from the Architecture. Google pushed this logic further with the Generic Kernel Image production. There is a Linux core and shared bulk for which vendor make their own services. These services link Android to the different elements of the Architecture and are to be maintained by the vendor. Google, to deliver updates, does not have to wait for the vendor latest version, to build a complete new image, and can directly feed its update at its own pace. An extensive cross-platform analysis of security patches in 682 open-source projects [59] computes that half of the vulnerabilities lifespan exceeds 14 months in their respective projects. If authors could not find a correlation between severity and lifespan, they still provide that the average lifespan is 5 years and that 6.5% of the vulnerabilities are present for less than 10 days. Over a single version, as for Debian Wheezy [60], an analysis shows no maturity of a LTS version. Back to Android, regarding its Linux kernel and Qualcomm components, these are usually exposed for one complete month more between the moment a patch is made available and before the update can reach a phone [61]. Those analyses can heavily benefit from tools permitting the generation of vulnerability datasets like data7 [62]. Data7 gathers CVE related intel from the NIST's NVD website, and another resource to generate one single database.

VIII. Conclusion and Future Work

In this work, we have studied how the Android system implements and maintains the JCL and what the security implications are. Our observations are that Android mostly relies on the OpenJDK external dependency for its JCL implementation. The complexity of the Android JCL implementation increases

[19]https://developer.android.com/ndk/downloads/revision_history

[20]https://developer.android.com/docs/

[21]Section Security Changes: https://developer.android.com/about/versions/jelly-bean#android-4.2

over time, because of the increase in the number of OpenJDK classes imported in Android, the fragmentation in terms of number of different OpenJDK versions used in Android, and the increase in the distance to OpenJDK's JCL classes. At the moment, OpenJDK vulnerabilities which make their way into the Android code base are not actively patched. Most of them affect the availability of the Android system. We show that they can be exploited by developing a proof-of-concept. We highlight that public information such as OpenJDK CVEs should be taken into account in the Android development pipeline to reduce the number of vulnerabilities imported in Android from external dependencies.

In future work, we plan to investigate the security impact of the fragmentation of the OpenJDK versions in Android and to understand the divergence between OpenJDK and Android.

IX. Acknowledgements

This work has been supported by Kempestiftelserna and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 617–632, 2014.

[2] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 627–638, 2011.

[3] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 217–228, 2012.

[4] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Revisiting android permission specification analysis," in *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, (USA), p. 1101–1118, USENIX Association, 2016.

[5] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *2013 IEEE International Conference on Software Maintenance*, pp. 70–79, IEEE, 2013.

[6] M. Kechagia, M. Fragkoulis, P. Louridas, and D. Spinellis, "The exception handling riddle: An empirical study on the android api," *Journal of Systems and Software*, vol. 142, pp. 248–270, 2018.

[7] H. S. Borges and M. T. Valente, "Mining usage patterns for the android api," *PeerJ Computer Science*, vol. 1, p. e12, 2015.

[8] R. Spreitzer, G. Palfinger, and S. Mangard, "Scandroid: Automated side-channel analysis of android apis," in *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pp. 224–235, 2018.

[9] M. A. Lemley and P. Samuelson, "Interfaces and interoperability after google v. oracle," *Tex. L. Rev.*, vol. 100, p. 1, 2021.

[10] O. H. Alliance, "Industry leaders announce open platform for mobile devices," 2007. Blog post.

[11] J. Brodkin, "Sun wanted up to $50 million from google for java license, schmidt says," *Ars Technica*, 2012.

[12] B. Sutor, "Ibm joins the openjdk community, will help unify open source java efforts," *Archived from the original on*, vol. 18, 2010.

[13] E. R. Wognsen, H. S. Karlsen, M. C. Olesen, and R. R. Hansen, "Formalisation and analysis of dalvik bytecode," *Science of Computer Programming*, vol. 92, pp. 25–55, 2014. Special issue on Bytecode 2012.

[14] S. Mullan, "Jep 411: Deprecate the security manager for removal," 2021. https://openjdk.org/jeps/411.

[15] P. Holzinger, S. Triller, A. Bartel, and E. Bodden, "An in-depth study of more than ten years of java exploitation," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), p. 779–790, Association for Computing Machinery, 2016.

[16] A. Bartel and J. Doe, "Twenty years of escaping the java sandbox," *Phrack*, 2018.

[17] J. Oliver, C. Cheng, and Y. Chen, "Tlsh – a locality sensitive hash," in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pp. 7–13, 2013.

[18] P. J. Landin, "The mechanical evaluation of expressions," *Comput. J.*, vol. 6, pp. 308–320, 1964.

[19] A. D. Blog, "Here comes treble modular base for android," 2017. https://web.archive.org/web/20170617040503/https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html.

[20] A. D. Blog, "Fresher os with projects treble and mainline," 2019. https://android-developers.googleblog.com/2019/05/fresher-os-with-projects-treble-and-mainline.html.

[21] A. Documentation, "Android common kernels." https://source.android.com/docs/core/architecture/kernel/android-common.

[22] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, feb 2019.

[23] B. T. Kurotsuchi, "The wonders of java object serialization," *XRDS: Crossroads, The ACM Magazine for Students*, vol. 4, no. 2, pp. 3–8, 1997.

[24] H. Osman, A. Chis, C. Corrodi, M. Ghafari, and O. Nierstrasz, "Exception evolution in long-lived java systems," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 302–311, 2017.

[25] P. Pufek, H. Grgić, and B. Mihaljević, "Analysis of garbage collection algorithms and memory management in java," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1677–1682, 2019.

[26] K. Jezek, J. Dietrich, and P. Brada, "How java apis break – an empirical study," *Information and Software Technology*, vol. 65, pp. 129–146, 2015.

[27] N. Meng, S. Nagy, D. D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), p. 372–383, Association for Computing Machinery, 2018.

[28] F. Long, "Software vulnerabilities in java," tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2005.

[29] I. Sayar, A. Bartel, E. Bodden, and Y. Le Traon, "An in-depth study of java deserialization remote-code execution exploits and vulnerabilities," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, feb 2023.

[30] P. Holzinger and E. Bodden, "A systematic hardening of java's information hiding," in *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems*, ASSS '21, (New York, NY, USA), p. 11–22, Association for Computing Machinery, 2021.

[31] Y. Chen, T. Su, and Z. Su, "Deep differential testing of jvm implementations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1257–1268, 2019.

[32] W. Bonnaventure, A. Khanfir, A. Bartel, M. Papadakis, and Y. Le Traon, "Confuzzion: A java virtual machine fuzzer for type confusion vulnerabilities," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pp. 586–597, IEEE, 2021.

[33] R. Coppola, L. Ardito, and M. Torchiano, "Characterizing the transition to kotlin of android apps: A study on f-droid, play store, and github," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, WAMA 2019, (New York, NY, USA), p. 8–14, Association for Computing Machinery, 2019.

[34] M. Martinez and B. Gois Mateus, "Why did developers migrate android applications from java to kotlin?," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4521–4534, 2022.

[35] L. Ardito, R. Coppola, G. Malnati, and M. Torchiano, "Effectiveness of kotlin vs. java in android app development tasks," *Information and Software Technology*, vol. 127, p. 106374, 2020.

[36] A. Mazuera-Rozo, C. Escobar-Velásquez, J. Espitia-Acero, D. Vega-Guzmán, C. Trubiani, M. Linares-Vásquez, and G. Bavota, "Taxonomy of security weaknesses in java and kotlin android apps," *Journal of Systems and Software*, vol. 187, p. 111233, 2022.

[37] C.-M. Lin, J.-H. Lin, C.-R. Dow, and C.-M. Wen, "Benchmark dalvik and native code for android system," in *2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*, pp. 320–323, 2011.

[38] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein, "Borrowing your enemy's arrows: The case of code reuse in android via direct inter-app code invocation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, (New York, NY, USA), p. 939–951, Association for Computing Machinery, 2020.

[39] M. D. Syer, M. Nagappan, B. Adams, and A. E. Hassan, "Studying the relationship between source code quality and mobile platform dependence," *Software Quality Journal*, vol. 23, p. 485–508, sep 2015.

[40] P. Liu, L. Li, Y. Yan, M. Fazzini, and J. Grundy, "Identifying and characterizing silently-evolved methods in the android api," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 308–317, 2021.

[41] G. Yang, J. Jones, A. Moninger, and M. Che, "How do android operating system updates impact apps?," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, (New York, NY, USA), p. 156–160, Association for Computing Machinery, 2018.

[42] 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.

[43] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), p. 627–638, Association for Computing Machinery, 2011.

[44] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, (New York, NY, USA), Association for Computing Machinery, 2012.

[45] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android.," in *Ndss*, vol. 310, pp. 20–38, 2013.

[46] A. Armando, A. Merlo, and L. Verderame, "An empirical evaluation of the android security framework," in *Security and Privacy Protection in Information Processing Systems* (L. J. Janczewski, H. B. Wolfe, and S. Shenoi, eds.), (Berlin, Heidelberg), pp. 176–189, Springer Berlin Heidelberg, 2013.

[47] D. Cotroneo, A. K. Iannillo, R. Natella, and R. Pietrantuono, "A comprehensive study on software aging across android versions and vendors," in *Empirical Software Engineering*, 2021.

[48] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, jun 2014.

[49] M. Backes, S. Bugiel, O. Schranz, P. Von Styp-Rekowsky, and S. Weisgerber, "Artist: The android runtime instrumentation and security toolkit," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 481–495, 2017.

[50] J. Huang, O. Schranz, S. Bugiel, and M. Backes, "The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android," in *Computer and Communications Security (CCS)*, ACM, 2017.

[51] O. Schranz, S. Weisgerber, E. Derr, M. Backes, and S. Bugiel, "Towards a principled approach for dynamic analysis of android's middleware," *arXiv preprint arXiv:2110.05619*, 2021.

[52] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: Fuzzing android native system services via automated interface analysis," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 307–323, USENIX Association, Aug. 2020.

[53] D. Cotroneo, A. K. Iannillo, and R. Natella, "Evolutionary fuzzing of android os vendor system services," *Empirical Software Engineering*, vol. 24, pp. 3630–3658, 2019.

[54] M. Zalewski, "American fuzzy lop." http://lcamtuf.coredump.cx/afl/, 2017.

[55] G. Gong, "Fuzzing android system services by binder call to escalate privilege," *BlackHat USA*, vol. 2015, 2015.

[56] Y. Aafer, G. Tao, J. Huang, X. Zhang, and N. Li, "Precise android api protection mapping derivation and reasoning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), p. 1151–1164, Association for Computing Machinery, 2018.

[57] H. Cavusoglu, H. Cavusoglu, and J. Zhang, "Security patch management: Share the burden or share the damage?," *Management Science*, vol. 54, no. 4, pp. 657–670, 2008.

[58] A. Arora, R. Telang, and H. Xu, "Optimal policy for software vulnerability disclosure," *Management Science*, vol. 54, no. 4, pp. 642–656, 2008.

[59] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), p. 2201–2215, Association for Computing Machinery, 2017.

[60] N. Alexopoulos, S. M. Habib, S. Schulz, and M. Mühlhäuser, "The tip of the iceberg: On the merits of finding security bugs," *ACM Trans. Priv. Secur.*, vol. 24, sep 2020.

[61] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags, "An empirical study of android security bulletins in different vendors," in *Proceedings of The Web Conference 2020*, WWW '20, (New York, NY, USA), p. 3063–3069, Association for Computing Machinery, 2020.

[62] M. Jimenez, Y. Le Traon, and M. Papadakis, "[engineering paper] enabling the continuous analysis of security vulnerabilities with vuldata7," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 56–61, 2018.