Research paper

A risk estimation study of native code vulnerabilities in Android applications

Silvia Lucia Sanna [*], **Diego Soi**, **Davide Maiorca**, **Giorgio Fumera**, **Giorgio Giacinto**

Department of Electrical and Electronic Engineering, University of Cagliari, 09123 Piazza d'Armi, Italy

*Corresponding author. E-mail: silvial.sanna@unica.it

Abstract

Android is the most used operating system (OS) worldwide for mobile devices, with hundreds of thousands of apps downloaded daily. Although these apps are primarily written in Java and Kotlin, advanced functionalities such as graphics or cryptography are provided through native C/C++ libraries. These libraries can be affected by common vulnerabilities in C/C++ code (e.g. memory errors such as buffer overflow), through which attackers can read/modify data or execute arbitrary code. The detection and assessment of vulnerabilities in Android native code have only been recently explored by previous research work. In this paper, we propose a fast risk-based approach that provides a risk score related to the native part of an Android application. In this way, before an app is released, the developer can check whether the app may contain vulnerabilities in the native code and, whether present, patch them to publish a more secure application. To this end, we first use fast regular expressions to detect library versions and possible vulnerable functions. Then, we apply scores extracted from a vulnerability database to the analyzed application, thus obtaining a risk score representative of the whole app. We demonstrate the validity of our approach by performing a large-scale analysis on more than 100 000 applications (but only 40% contained native code) and 15 popular libraries carrying known vulnerabilities. The attained results show that many applications contain well-known vulnerabilities that miscreants can potentially exploit, posing serious concerns about the security of the whole Android applications landscape.

Keywords: vulnerability detection; Android app; native code

Introduction

The usage of mobile devices is increasingly growing due to their continuous advancements that allow people to carry out very different tasks, from surfing the internet to accessing banking or medical accounts. Smartphones are also extensively employed as multimedia devices (e.g. to watch movies or play games) and as aids for payment authentication and Public Administration services. Unfortunately, this variety of usage allows attackers to exploit vulnerabilities (by resorting to, e.g. phishing emails and messages or by exploiting memory errors) to take control of the target devices.

Among the various operating systems available for mobile devices, Android is the most used worldwide [1], and many of its applications can feature hundreds of millions of downloads. These apps often need to interact with native activities and components (e.g. camera and microphone) available through native code (typically C/C++) implementation, which may be written from scratch or taken from third-party libraries such as Libpng and OpenCV. For brevity, we refer to native third-party libraries as *products*. In most cases, developers use publicly available libraries such as Libpng (for image management) by importing them into their projects. As native libraries are written in memory-unsafe languages, they can suffer from typical vulnerabilities caused by wrong source code programming or design. Improperly managing pointers, arrays, and API calls can lead to overflow attacks or other vulnerabilities. A simple example of possible memory errors is *buffer overflow*, which allows an attacker to send an input whose size is larger than required, thus writing data outside bounds and causing unpredictable behaviors. Exploiting vulnerabilities in native libraries can affect the functionality of the whole application, leading to some data exposure or, in the worst cases, to the loss of control of the device. For this reason, it is essential to manage the security of the used libraries when developing an Android ap-

plication. Previous research works have only recently pointed out the need for better native code safety and vulnerability analysis [2].

However, finding and analyzing vulnerabilities is a very time- and resource-consuming task requiring in-depth static and dynamic analysis of the native layer and its interaction with the Java/Kotlin code [2]. Recent works also showed several validation problems related to the effective *reachability* of vulnerable functions [3]. These issues may discourage analyzing the native layer security in their apps, thus often overlooking even well-known issues of public products. We propose a probabilistic approach that vulnerability researchers can use to have a first basic idea of the vulnerability to be checked manually. Our vulnerability detection on Android native code can be included in the process of producing and maintaining the software bill of materials (SBOM), a detailed inventory of software components and their ingredients essential in software security and supply chain risk management (as described by the American Cybersecurity and Infrastructure Security Agency [4]). Different organizations worked on that, such as NIST [5], who released in February 2022 guidelines to be followed by developers and companies as a means of cyber-attack prevention. In fact, SBOM has been introduced to provide guidance on the level of risk associated with the software, whether stand-alone or integrated into systems (such as in the case of Android native code). SBOM defines the most dangerous vulnerabilities and gives a global risk indication of the software vulnerabilities, stating the components with a greater likelihood of being affected (as in our methodology).

This paper proposes an alternative strategy for native code vulnerability identification that does not involve resource-heavy analyzes but leverages on *public knowledge of known issues*. The idea is to yield a quick, lightweight approach that gives an idea of an application's possible *known risks* to take immediate actions to address them. This is done through *a risk assessment algorithm* that leverages a combination of quick code analysis and public domain knowledge to provide a score of possible dangerousness of the application based on the vulnerabilities found.

More specifically, our contributions can be summarized as follows: (i) we propose a *minimal complexity* native code analysis strategy oriented to the search for known vulnerabilities and issues by leveraging *public domain knowledge*; (ii) we define a *risk assessment algorithm* that provides a dangerousness score that can aid security researchers to take immediate actions to patch the analyzed applications; and (iii) we evaluate our methodology through a large-scale analysis on 100 000 APKs taken from the widespread application repository Androzoo[1], but results are focused on 38 348 apks, which are those using at least one native library. To the best of our knowledge, no risk assessment algorithm or methodology has been published for vulnerabilities in the native code. The results attained in this paper demonstrate that a risk-based approach can be strongly beneficial in swiftly assessing vulnerabilities in Android applications, thus addressing this problem by working on their early detection and prevention.

The remainder of this paper is structured as follows: section "Technical background" presents a technical background about Android applications structure and vulnerabilities. Previous research is illustrated in section "Related works," while the applied methodology is presented in section "Methodology and implementation details." Results are reported in section "Results." Finally, section "Summary, limitations, and future works" discusses the lim-

its and the future works that may be conducted to improve this work.

Technical background

Before introducing our methodology, some concepts need a brief explanation to provide the reader with basic knowledge about the core elements of Android applications and native code.

Advanced RISC machine

ARM, the acronym for **Advanced RISC Machine**[2], is the hardware architecture on which Android operating system (OS) and apps are executed. It is commonly implemented in embedded systems, where developers design and sell the processor's architecture to vendors such as Samsung, Lenovo, and Oppo. ARM is based on **RISC** (i.e. reduced instruction set computer), an architecture with a smaller instruction set than x86/64 but with more general-purpose registers and a load/store mechanism. As an example, to modify a value of a register, it is required to move the value to the register (with the instruction load), make the desired arithmetic operations, and save it back to memory (with the instruction store).

Regarding arithmetic operations, the architecture reduces branch complexity and number of instructions by supporting conditional execution (less than equal, greater than equal) and barrel instructions (shift and rotation). ARM is also useful for implementing co-processors by allowing the execution of different tasks to different cores of one processor. Hence, the program execution time is inversely proportional to the number of cores.

Android OS

The proprietary open-source Android OS [6], published by Google in the early 2000s, is the operating system running on ARM hardware and on which apps are built. Android mostly features six main layers [7]: (i) **Android system apps**, featuring apps for standard activities (e.g. SMS, calendar, emails); (ii) **Java API framework** containing Android APIs to make different software components communicate with each other; (iii) **native libraries and core system services** written in C/C++ to manage activities and interact with physical device components; (iv) **Android runtime** to manage runtime for executing Android apps since Android 5.0; (v) **hardware abstraction layer** (HAL), which is a software-hardware interaction layer that employs specific hardware interface description language (HIDL), allowing detachment between OS and drivers (autonomous upgrade); and (vi) **Linux kernel**, based on an upgraded version of Linux kernel to such platform.

An interesting characteristic of Android OS is the permission level. In *low-level* mode, users and groups can access file systems and specific resources. Conversely, permissions are restricted in *high-level* mode, and apps are installed.

Android applications and framework layers are executed in Android runtime (ART). ART is the runtime system that executes Dalvik Executable format and Dalvik bytecode. Since Android 5.0 Lollipop, it replaces the **Dalvik virtual machine** (DVM), a register and Java-based virtual machine designed to give an efficient abstraction layer to the OS. With the DVM, developers had to partially compile the app, while the DVM did the other parts at runtime. Instead, in ART, Dalvik bytecode is compiled for ARM assembly dur-

---

1 https://androzoo.uni.lu/

2 https://www.arm.com/

ing installation, and the app can run directly by executing machine instructions.

Android apps

As described in the previous paragraph, Android apps [8] work in the Android application layer, which allows developers to extend OS functionalities without altering lower levels. To build an Android app, developers write source code in Java/Kotlin that is compiled to .class files. Then, these are translated to Dalvik bytecode and compiled to a single .dex file (Dalvik executable), which is optimized, loaded, and interpreted by the DVM when the app is run. Once the app is compiled, it is packed as an .APK file, a sort of zip archive containing all files needed for its execution, and structured as follows:

- **AndroidManifest.xml**, a file containing components and permissions needed by the app;
- **classes.dex** containing the assembled source code (i.e. list of classes, methods, and bytecode);
- **res**, a directory containing all elements related to the visual presentation;
- **assets**, a directory containing external resources used by the app while under execution;
- **META-INF**, a directory with the files used by the Java platform to interpret and configure the app;
- **lib**, a directory containing all platform-dependent native libraries written in C/C++ and compiled in the different **ABIs** (application binary interface[3]) found in different sub-directories.

Each app is also made of four different active asynchronous component types. These are the *activities*, entry points for user interaction; *services*, general-purpose entry points to keep the app running in the background; *broadcast receivers*, intercommunication activities between the apps and the system; and *content providers*, whose aim is to allow apps to store and share data in the file system.

Native libraries

As explained before, developers use native code to interact with native components and hardware. Moreover, developers can use Android NDK (Native Developer Kit [9]) to import third-party native libraries without re-implementing them. These libraries are typically compiled C/C++ code with two kinds of extensions: .so, which can be defined as native shared libraries; and .a, namely native static libraries, which are those linked to others.

Once compiled, the libraries are **ELF** (executable and linkable format) files like the ones resulting from compiling a C code for Linux. The three most essential headers are (i) *ELF header*, with the *magic number* to recognize the file format, compilation architecture, version, and information about sections; (ii) *program header* to create a process image; and (iii) section header containing all the file's sections (.bss, .data, .text, etc.).

The analysis of an ELF can be performed with several command-line tools (e.g. *readelf* and *elfdump*), reverse engineering software such as IDAPro and Ghidra, or Python libraries like pwntools, which were employed in this work as explained later in section "Methodology and implementation details."

Common vulnerability exposure

Once a vulnerability is found in any application or hardware, it is often disclosed in a public dataset following the **CVE standard** [10] with a name featuring the form *CVE-DisclosureYear-ProgressiveNumberForThatYear*. Each database entry contains the product (i.e. an extensive library known commercially, such as Libpng or OpenCV), the affected product vendors, the publication date, and a human-readable description. The latter is very important, as it contains information about the vulnerability, such as the name of the vulnerable product (with the version) and the name of the function that allows the attack to be carried out. Sometimes, the attack and procedure to patch the vulnerability are also described. Each CVE is also quantified with a score according to **CVSS** (Common Vulnerability Score System) [11]. The latter defines three metric groups:

- *Base metric*, a constant severity value over time and across the user platform. It is composed of the **Exploitability metric**, which expresses the ease and technical levels required to exploit the vulnerability, and the **Impact metric** that quantifies the damage due to a successful exploit;
- *Temporal metric*, a severity value that considers vulnerability's changes over time but is constant across the user platform;
- *Environmental metric*, which reflects severity scores depending on the user's environment, possibly considering the presence of defence systems and security controls to mitigate the consequences of an attack.

At the moment of writing this paper, different vulnerabilities have been published for Android: about 6305 vulnerabilities in Android OS by Google, more than 100 by Samsung; only 2 for Motorola; only 3 vulnerabilities regarding Android hardware. Additionally, some vulnerabilities directly concern Android applications (the majority of them regarding Java management). Notably, vulnerabilities in Android native code cannot be addressed easily and directly as the native code is mostly embedded in other third-party libraries (see later sections of this work).

Up to now, there are different public databases containing CVEs, such as Mitre CVE [12], CVE Details [13], and National Vulnerability Database by NIST (NVD) [14]. Most public databases about vulnerabilities are derived from Mitre, but they add some details and more technical information.

Risk assessment

In this work, we developed a risk assessment algorithm for vulnerabilities in the native code of Android applications. We now provide a brief introduction to the topic.

Risk assessment includes a set of techniques and methods to determine the risk of an asset in a specific scenario. It is not only used in cybersecurity, but it is a general concept that can be applied to almost every field where undesired events could affect and damage the system. Given a specific scenario and a potential threat to an asset, we have to evaluate the *likelihood* of the threat damaging the asset[4] in a quantitative (with numbers and specific metric measure), semi-quantitative (with numbers without a specific metric measure), or qualitative (with specific terms) way. The threat is often considered a deliberate attack, depending on the attacker's capability and the infrastructure protection mechanisms. The damage, of course, is the loss of infrastructure when the attack is successful. Technically, the risk assessment procedure involves tests such as penetration tests

---

3   An application binary interface (ABI) is an interface between the operating system and its applications. Each ABI is defined in Android by the combination of CPU and instruction set because each device uses its own.

4   In cybersecurity, the asset is the technical infrastructure we have to protect from cybercriminals.

and simulation attacks where the company assesses the likelihood of being attacked. In general, after a risk assessment study, the company takes countermeasures to improve the protection of the assets. Different standards have been published about risk assessment, such as ISO 27005:2008 [15] (the one we used in this work) and NIST 800-30.

Related works

Identifying Android native code vulnerabilities is one of the emerging research fields of cybersecurity, and as far as we know, very few works have been published on this topic. The prominent one is *Librarian* [2], where the authors studied vulnerabilities in the top 200 Android apps downloaded from the Google Play Store from September 2013 to May 2020. They studied a new algorithm called *bin2sim*, capable of extracting six features from each ELF in the app. Additionally, by applying binary similarities techniques, the tool correctly identifies different libraries and versions, implementing a whitelist approach for vulnerability detection. Despite the accuracy of this tool, their methodology is very heavy from the point of view of computational complexity. Moreover, they only say whether the APK and the native library are vulnerable or not without assigning a risk score. Since it is one of the most recent works on this topic, we decided to compare our methodology with their results, as we also based the product selection on their criteria and considered the ± 2 years time elapsed to patch a vulnerability. As described in section "Comparison with Librarian results," we had to find a way to compare our risk score on their dataset with their result because of the lack of score in their results.

Although the early versions of the CVSS score have not been designed as a metric for risk estimation, over the years, the metric evolved to provide a reliable measure to evaluate the impact of vulnerabilities and exploits. One of the seminal works that pointed out the weaknesses of the previous versions of CVSS was the paper by Allodi and Massacci [16], who in 2014 criticized the usage of pure CVSS base score without considering the presence of exploits in the wild for a given vulnerability. The authors proposed a novel way to include known attacks by merging CVEs published in NVD and exploits released on exploit-db (repository of computer software exploits and exploitable vulnerabilities), eits (black markets exploits), and sym (vulnerabilities exploited in the wild). By employing this methodology, they could assess the limitations of the CVSS score first version and reduce the risk sensitivity according to the known exploits of 45%. The third version of CVSS (v3.1, the one used in this work for the experimental part) is more consistent. A brand new version has been released during this writing: CVSS v4.0 reinforces the concept of CVSS as not just a mere base score, as it considers threats, environments, attack requirements, user interactions, and other metrics focused on a more real CVSS value, as suggested by Allodi and Massacci.

Other works addressed vulnerability detection. For example, Alves et al. [17] studied the correlation between software metrics and software vulnerabilities. The authors claim that metrics exist to identify bad software, which is also harder to verify and maintain, with unnoticed or inadvertently introduced vulnerabilities. The authors compiled 5750 vulnerabilities from Linux Kernel, Mozilla, Xen Hypervisor, httpd, and glibc. Analyzing 2875 security patches, they distinguished vulnerable and safe functions. The results emphasize early vulnerability management and the need for developers to use multiple metrics for predicting code vulnerabilities. Even Madeiros et al. [18] addressed this topic with a study on software metrics useful to de-

tect security vulnerabilities in software development. They analyzed various software metrics, such as complexity and coupling metrics, as well as other structural quality indicators, and identified patterns and correlations indicating the presence of security vulnerabilities. The authors established a correlation between specific project-level metrics and the number of vulnerabilities present in the software systems. They also found a specific group of discriminative metrics different across the software systems but present in all of them and valuable to distinguish between vulnerable and non-vulnerable code. The software metrics were identified using a genetics algorithm and a random forest classifier. Instead, in [19], a method called MVP (matching vulnerabilities and patches) has been presented to detect vulnerabilities using patch-enhanced vulnerability signatures with low false positive and false negative rates. This methodology can distinguish between already patched vulnerabilities and generate accurate vulnerability and patch signatures to improve vulnerability detection accuracy. Du et al. developed LEOPARD [20], a framework in a lightweight approach to help security experts detect potentially vulnerable functions in a code base without prior knowledge of the known vulnerabilities. Leopard combines complexity and vulnerability metrics to identify potentially vulnerable functions, providing a more comprehensive vulnerability assessment. The vulnerability is detected at all levels of complexity without missing the low-complex ones. For this purpose, the authors used a binning-and-ranking approach, where functions are grouped into bins based on complexity metrics and then ranked within each bin using vulnerability metrics. The framework covers a substantial portion of vulnerable functions identified, while only a fraction is flagged as potentially vulnerable, outperforming machine learning and static analysis methods.

Another interesting problem regarding vulnerability detection is **reachability**, i.e. analyzing whether or not a vulnerable function is called in the app during its execution. Borzachiello et al. proposed DROIDREACH [3], a tool to detect reachable APIs using heuristic and symbolic execution. They were able to represent all possible paths a function may take within the *inter-procedural control-flow graph* (ICFG), whose aim is to encode all paths starting from an application entry point. Due to the complex methodology introduced in their work and the high computational complexity, we did not implement it in our work but considered this case in the risk methodology. Instead, we highlighted the imported library issue in our work as they did.

Recently, Ruggia et al. [21] developed a new methodology to reverse engineer Android apps, focusing on identifying suspicious patterns related to native components. They used suspicious tags to train a machine learning algorithm for binary classification. In particular, they developed a static tool that analyzes the code blocks responsible for suspicious behaviors in detail. This work demonstrates the use of native code in malicious Android applications so that the analysis is more complex and the maliciousness is better achieved.

One of the first studies on Android native code exploits was proposed in 2013 by Fedler et al. [22]. They introduced different techniques to provide various levels of protection against all known local root exploits without affecting the user experience. Their mitigation reduced the exploitability of Android devices. In those years, very few Android applications used native code, and their approach was unsuccessful in exploiting and targeting flaws in the DVM. Nowadays, more applications use native code, and Android architecture has changed. It is also worth noting that even popular tools for Android APK vulnerability detection, such as MobSF [23] and Qark [24], and SEBASTiAn [25] do not look for vulnerabilities in the native code, even whether they are good vulnerability detection tools.

Fuzzing is another technique to detect vulnerabilities, and recently, it has also been used in Android native code. One of the most popular fuzzers is AFL++, which has been adapted in Frida mode [26] to interact and fuzz Android applications: detect the vulnerability in the C/C++ code and also check the interaction with the Java/Kotlin code. Other tools have been released, such as Android-AFL [27] and Libfuzzer [28]. All these tools are resource-consuming and sometimes could be more efficient in detecting. Different works [29] are focusing on fuzzing Android native components, but, as far as we know at the moment of writing this paper, none of them focuses on fuzzing Android application native code.

Android CVE analysis has been studied by Brant et al. in [30] with a focus on the Android security bulletin within the last six years from 2022. According to them, to have more secure Android systems, security bulletin updates must be designed with specific tests and improve code coverage of patched files. Only 13% of security bulletin updates contain fixed test files for that particular update, and among these, only 42.8% have full patch coverage. Even if the percentage is still low, this is an interesting result, meaning that the community is beginning to address Android security and vulnerability detection.

LibRadar [31] demonstrated that a whitelist approach is inefficient because package names can be modified in many ways. For this reason, they released a detection tool based on stable code features that are obfuscation-resilient, such as APIs, which are also obfuscation-resilient. We tried to use their approach, but at the time of this writing, the tool was no more accessible. Another interesting approach is the one proposed by Li et al. [32], which identifies libraries according to reference and inheritance relations between Java classes, methods, and other app metadata. Notably, a native library cannot be identified only according to Java interaction and inheritance, but specific C code syntaxes must be considered.

Other works such as *GoingNative* [33], *NativeGuard* [34], and *AppCage* [35] focused only on the isolation and secure sandboxing of native code in Android applications by running the app in a protected but unrealistic environment. These methodologies are fundamental but cannot be considered in a real-world scenario where customized sandboxes are rarely employed. On the contrary, *Ndroid* [36] and *DroidNative* [37] focused on data flow between Java and native code and native code control flow patterns, also for malware detection. Finally, *AdDetect* [38] is a framework for advertisement library detection using semantic analysis with machine learning and hierarchical clustering techniques. The interaction between native code and Java code is critical and must be considered, even if we are limited to C code vulnerability detection in our work.

As the "Introduction" mentions, most of these works feature high time and space computational complexity. For example, using machine learning approaches, an extensive dataset of samples is needed to train the model correctly, and the fine-tuning of the model can introduce additional complexities. On the other hand, the methodologies above in the literature have a good accuracy in the results. Instead, the methodology we propose in this paper needs very few resources; it is fast to execute even with a large-scale dataset and does not need any dataset on which to train the machine learning algorithms.

One of the scenarios for which this work has been developed is the SBOM, as explained in the "Introduction." This concept has been well explained by Zahan et al. [39], where they discuss the importance of SBOM in improving cybersecurity, highlighting its benefits and challenges. The work has been based on the Log4Shell vulnerability: a zero-day remote code execution vulnerability discovered in Apache Log4j that significantly impacted software organizations in December 2021 despite a few months earlier, EO (Executive Or-

der) 14028 [40] recognized SBOMs as a practice for enhancing software supply chain security, and NTIA released a report of minimum points to use SBOM in risk reduction. After the Log4Shell vulnerability, NIST recognized SBOM as one of the official practices organizations must follow to reduce cyber attacks and listed it in the first version of the Secure Software Development Framework. Moreover, the industries following SBOMs were able to identify the Log4Shell vulnerability rapidly and had a more effective response.

In the work, Zahan et al. [39] highlight the benefits of using SBOMs, which include risk management, vulnerability detection, supply chain transparency, proactive management of security risks, and effective response to cyber threats. As part of EO 14028, SBOMs can improve the nation's cybersecurity, but it still has to be standardized in the industry, and some challenges still must be solved, such as the standardization of data requirements, the enhancement of solid guidelines, and the practitioners' collaboration.

Methodology and implementation details

This section illustrates the methodology used for vulnerability detection in the native code of Android applications. First, we describe the creation of the purpose-specific database. Then, we detail the library extraction algorithm we developed to analyze the information from each APK studied. Lastly, the risk assessment algorithm is fully explained.

Given an APK, we extract the native library from its *lib* directory. As the compiled library is an ELF file, we need specific reverse engineering tools and techniques to extract data for vulnerability detection. Such data is the list of functions and the product name, along with the version to which the library belongs. Once we have this information, we can match this result with a database of known vulnerabilities. The database is *purpose-specific*, containing for each CVE a field with the affected vulnerable version and function. At the moment of this writing, there is no publicly available database for this purpose, and with this specific structure, a system can easily access and read it.

We need a list of *N* products to construct the database, which are those whose vulnerabilities we want to look for in the apps. The product matching process is a *whitelist approach* that assumes that none of the Android app's developers has changed the library name (keeping the real one employed during compilation time in the NDK). Once we have a match between the library under analysis and the database, we can assign a risk assessment score to the CVE found in the analyzed library. Our purpose is to give a risk value to each library (according to the risk of each CVE found) and, consequently, to each app to provide an alarm to developers and security researchers. The following sections will explain in more detail the methodology.

Building a custom CVE database

A CVE database is essential to check whether the extracted data from the analyzed library have been declared in a published vulnerability listing. We decided to employ a custom database to reduce the query time to a public online database. Hence, our local database is a dump of data contained in the online selected website of CVEs (e.g. NVD) and with fields organized according to the aim of the research. As explained in section "Common vulnerability exposure," the essential information about a vulnerability is the product's name with the version and the vulnerable functions of the library. All this information may be found inside the human-readable description. It does not follow a specific syntax such as "*In product P of version v.n, there is a vulnerable function F.*," so we need to employ natural language pro-

Table 1. CVE descriptions and corresponding vulnerable product version.

Description	Vulnerable product version
*In product P **before version v.n**, the F function can be used for buffer overflow.*	Every version <= v.n is vulnerable
*In product P **version v.n**, F function can be used for attack*	Version == v.n is vulnerable
*In product P **version v.n and after**, F function can lead to buffer overflow.*	Every version >= v.n is vulnerable
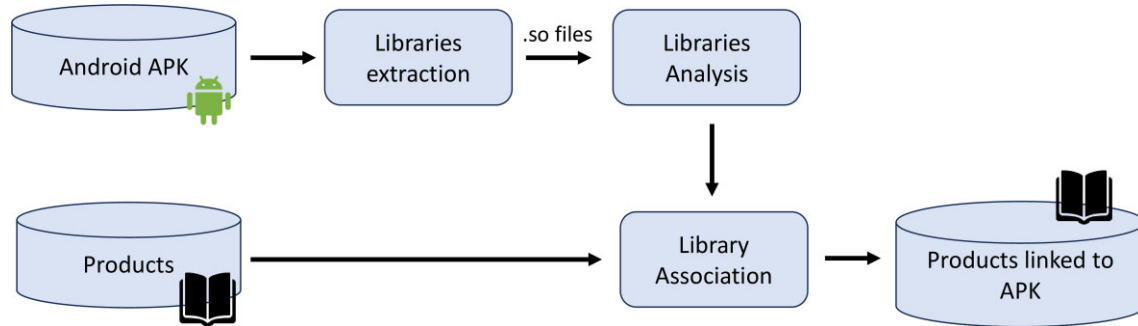
Figure 1. Workflow of the approach to extract, analyze, and associate native libraries, specifically in the use-case of vulnerability researchers.

cessing (NLP) techniques to process the description and extract the valuable data. Specifically, we employed Python nltk library[5], adapting it to our use case and syntax.

The **product name** is easily matched with one of the *N* selected products, and whether one is mentioned inside the description, we know that the CVE has been found in that specific product. This can be further confirmed by searching all vulnerabilities by product name in the public database.

The **version of the library** affected by the CVE is always a number that may come after the word *version*, the product name, or preceding words with the meaning of *after* or *before*. Some examples are described in Table 1: (i) whether there is a word with the meaning of *before* (i.e. before, prior, earlier, etc.), every product version lower than the one found in the description is vulnerable; (ii) whether there is a word with the meaning of *after* (i.e. after, following, successive), every product version higher than the one found in the description is vulnerable; and (iii) whether no word with the precedents meaning is found, the affected product version is only the one found in the description.

The last data to extract from the description is the **function name**. We rely our strategy on how programmers typically recognize function names and give names to functions. Then, by looking at some CVE descriptions, we noticed that the function is never declared as, e.g. *function F*, represented uniquely by its name. The function name usually contains specific symbols (i.e. _, ::, (), etc.). Moreover, whether the name is made up of more than one word, it is camel-cased. As an example, makeSum is made up of two words: make and Sum. In other cases, the function is not found, which means that the whole product version is vulnerable, but no description of the vulnerable function is provided.

This methodology was developed in iterative steps with manual cross-validation to check whether the algorithm worked correctly.

Library extraction, analysis, and association

The overall approach, subdivided into library extraction, analysis, and association, is depicted in Fig. 1. The presented methodology

can be used as it is by vulnerability researchers and adapted for developers to release a secure Android application.

Library extraction is the first step of the analysis part, and, as the name states, it consists of the extraction of the compiled library (ELF file) from the Android application. It is done by unzipping.so files from the lib directory of each APK. Indeed, libraries are compiled according to different ABIs and saved inside the application. We can choose to extract libraries only for specific architectures or to analyze libraries for all available ABIs. In this work, we considered all ELF files from armeabi-v7a directory and, if not available, looked for arm64-v8a or x86_64 as they represent the most popular architectures.

Library analysis is the part where we need to extract from the library the data for vulnerability detection. In particular, we need to know the list of functions in the library and the product name and the version to which the library belongs. This step can be done with different reverse engineering tools, such as Ghidra and its Python extension for automation. Typically, the product name and its version can be retrieved in the strings section (.rodata section), while the defined functions in the ELF file can be retrieved from the Symbol Table section. In this work, we used pwntools [41], a popular Python framework for binary analysis and exploitation. Specifically, we employed its ELF module, which allows the analysis of ELF files from which the strings and the list of functions are extracted. The version is taken from the result of the strings Linux utility.[6] At the same time, the functions are found inside the Symbol Table of the ELF without considering .got and .plt sections. A difficulty comes when binaries are stripped[7] where not all function names are available, or the names are unrecognizable.

Library association is the part where each ELF file is associated with at least one of the selected *N* products. This is a crucial task: whether we do not link each ELF file to its related product, we cannot determine whether the ELF file contains vulnerabilities, hence assigning a risk assessment score. Section "Related works" illustrates that different works used binary similarities techniques or machine

---

5 https://www.nltk.org/

6 strings tool in Linux is capable of retrieving all printable sequences of characters from the.rodata section of an ELF.

7 A stripped binary is a binary without some debugging symbols and so with a lack of data.

Table 2. Mapping from semi-quantitative to qualitative values of the product between threat and impact.

Semi-quantitative value	Qualitative value
$90 \div 100$	CRITICAL
$89 \div 70$	HIGH
$69 \div 40$	MEDIUM
$39 \div 0$	LOW

learning algorithms; however, we decided to apply a simple identification algorithm because we know that every product uses a clear and unique syntax in strings and function names. For example, in OpenCV, we can find strings like *General Configuration for OpenCV v.n* to declare the version and *xxxx_cv_xxxx* in the function names. For this reason, whether we find these syntaxes in strings and functions, we can link the analyzed ELF file to a product.

Some binaries can be linked to multiple products due to imported libraries. Indeed, in a compiled library, there can be traces of the *primary* library and the *secondary* libraries (i.e. the ones employed by the primary). We considered the ELF files belonging to all the retrieved products in this case, as a proper distinction between primary and secondary libraries can be hard to carry out in practice (an issue that has also been highlighted by Borzachiello et al. [3]).

When developers use our methodology, thanks to the library association step, they can find vulnerabilities in the *secondary* library (i.e. a library contained in the main library they are importing in the project as described here above) and take actions to mitigate them. Developers also use the library extraction and analysis part as we designed because when importing a third-party native library in an Android project for apk creation, Android Studio compiles the library, and to analyze it, developers have to extract and analyze the compiled ELF file as security analysts do. Instead, when developers want to check whether the native library contains vulnerabilities before importing them into the Android Studio project to create the application, they can immediately check the database, find a version with fewer vulnerabilities, and patch them.

Risk assessment algorithm

We developed a risk assessment algorithm to give a risk value to each library (and consequently to each app) to provide an alarm to security researchers. Even though a CVE is present in the analyzed libraries within an ELF, we are not 100% sure that it is exploitable due to stripped binaries, imported products, and without considering the reachability problem. Moreover, developers can rename the functions, patch their content, or use obfuscation techniques. For this reason, we can approach the problem in *probabilistic terms*, and we developed a semi-quantitative risk assessment algorithm.

According to **ISO 27005:2008** [15], the risk can be evaluated as the product between three factors:

$$\text{risk} = \text{threat} * \text{impact} * \text{vulnerability}. \qquad (1)$$

A **threat** corresponds to an action that negatively impacts the device. Hence, the threat factor can be associated with the ease with which an attack can be carried out. To quantify it, we can use the *CVSS exploitability* value since it has a similar definition, regardless of the attacker's capabilities. The **impact** is the damage caused to the system whether the vulnerability is exploited. It can be quantified with the CVSS impact value without considering the architecture of the victim device.

- **CRITICAL** when the CVE is present and surely exploitable.
- **HIGH** when a vulnerable library is found within the application, but we are uncertain about the CVE exploitability because the vulnerable API is not reachable, or we did not find the vulnerable function because the binary is stripped.
- **MEDIUM** when we can make the same assumptions of HIGH level for functions, but we cannot find the vulnerable version due to stripped binaries. Indeed, the library could be vulnerable because developers would be unaware of the dangers of function whether the CVE were released after the app's publication. Moreover, apps falling in this category feature a difference between their release and the CVE publication dates, which are inferior to two years. According to Librarian [2], two years is the time developers use to apply a patch and mitigate vulnerability effectiveness after its release. Hence, within this period, it is very likely that the library would be affected.
- **LOW** when we cannot state whether the CVE is present. So, we establish a small level of risk when the found version and functions are not associated with any CVE.
- **NONE** when no native library is found, or the analyzed ELF files do not belong to our N products.

Our study aims to establish a qualitative value of the risk. To do so, we have to rescale the semi-quantitative product between threat and impact (in a range between 0 and 100 as both of them have values between 0 and 10) into qualitative metrics, as detailed in Table 2. The values have been determined according to the CVSS 3.1 qualitative severity rating scale. Then, by applying equation (1), we evaluate the risk according to the matrix in Table 3. In this way, we have a qualitative risk assessment score to assign to each CVE present in each library of each Android application.

The purpose is to give a risk value to each library (and, consequently, to each app) and to provide swift alarms to security researchers. Once they are informed about the risks associated with the library or application, they can find a way to patch the vulnerability (e.g. upgrade the library to a non-vulnerable version, fix the library, etc.). To this aim, we assigned the highest score of the CVEs risk level to each library but also saved the affected CVEs in a log file. For example, whether we find five CVEs with a LOW level and one with a MEDIUM level, we assign a MEDIUM risk to the library for a more effective alarm. Another approach could be evaluating the average risk, giving the most representative risk: in the previous example, the result should be LOW, but in this way, we would underestimate the risk, which is far from our purpose. The same weighted approach has been adopted for the application risk level attribution.

Results

In this section, we illustrate the results of a large-scale analysis conducted on 100 000 APKs from Androzoo. Additionally, to prove the efficiency of our methodology and risk algorithm and to make a comparison between our approach and Libriarian [2], we applied the approach to 32 apps from the published dataset [42] by Almanee et al.

Since the Librarian dataset employs apps released between September 2013 and May 2020, we downloaded the updated versions of such apps (February 2023) and whether the vulnerability risk changed in this amount of time.

Table 3. Risk matrix to determine the qualitative risk value.

Threat*impact\vulnerability	None	Low	Medium	High	Critical
Critical	Medium	High	High	Critical	Critical
High	Medium	Medium	High	High	Critical
Medium	Low	Medium	Medium	High	High
Low	Low	Low	Medium	Medium	High

Table 4. Fifteen selected products to perform the analysis, a brief description of their purpose, the number of released CVEs in the last 5 years, and the percentage of apps in our dataset that contain these products.

Product	Description	# CVE	% Dataset
OpenCV	Real-time computer vision	34	5
OpenSSL	Secure communication over network	232	1
FFmpeg	Manage multimedia files (audio–video)	407	10
Libav[8]	FFmpeg fork to manage multimedia files	106	5
Sqlite 3	Database engine	54	8
LibWebp	Alternative to PNG, JPEG, GIF	14	5
Libpng	Handle PNG images	44	5
Libjpeg-turbo	Handle JPEG image format	1	16
Lua	Lua language interaction	15	3
Mono	Create compatible tools with Framework.NET	20	11
Folly	Core library components used by Facebook	3	11
Hermes	Fast startup of React Native apps	20	10
React-Native	Use React framework in applications	1	77
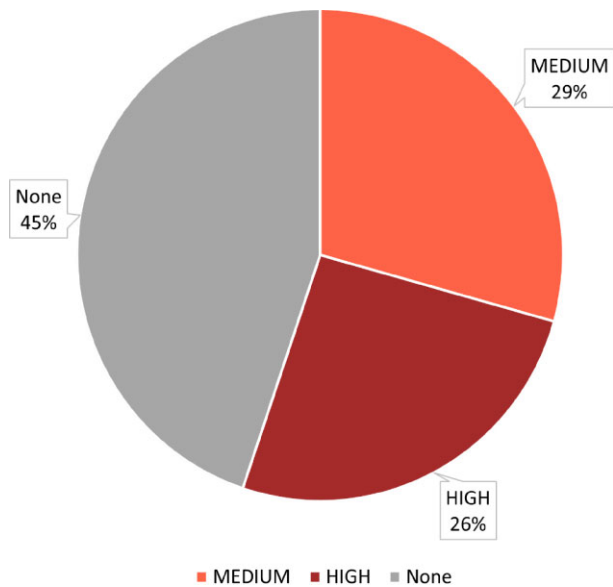
Figure 2. The pie chart shows the percentage of apps for which a risk level has been computed by only identifying 15 products. The NONE value means that the found native code does not belong to any of our 15 selected products, but it can have vulnerabilities related to other libraries.

Dataset and products

To apply the study to a large-scale dataset, we downloaded the first 100 000 applications found in the Androzoo collection, which is a popular dataset developed by the University of Luxembourg with about 23 million APKs dumped from different markets and years, also analyzed by various anti-malware engines. Among the 100 000 applications, we selected only 38 348, which contained native code (see GitHub repository[8]). In particular, since this study started in September 2021, we downloaded the samples considering the list of apps from Androzoo csv.

Additionally, we built a dataset of products, i.e. a set of N popular native libraries within the downloaded samples. This is a necessary step to associate each ELF file with at least one product, as expressed in section "Library extraction, analysis, and association." Due to time and space constraints, we selected the most representative libraries within the 38 384 APKs, considering the number of published vulnerabilities (CVE) for those libraries and the representativeness within the app chosen inside the large-scale dataset. So, in a similar fashion to what was made by Almanee et al. [2], given the 38 384 APKs, we made a list of the native libraries' names, associated them to their own products, and computed statistics of the products considering the number of the published known vulnerabilities and their percentage of diffusion in the dataset (considering the top products to be selected). At the end, we had a dataset of 15 products: OpenCV, OpenSSL, FFmpeg, Libavcodec, Libavformat, Libswresample, Sqlite 3, LibWebp, Libpng, Libjpeg-turbo, Lua, Mono, Folly, Hermes, React-Native.

Large-scale analysis results

Out of the 100 000 downloaded APKs, about 40% of them contained native code. In particular, there are 38 384 APKs with at least one native library and a total of 225 638 ELF files. Among these, 44 225 belong to at least one of our 15 products.

---

8   https://github.com/slsanna/Android_Native_Code_Vulnerability
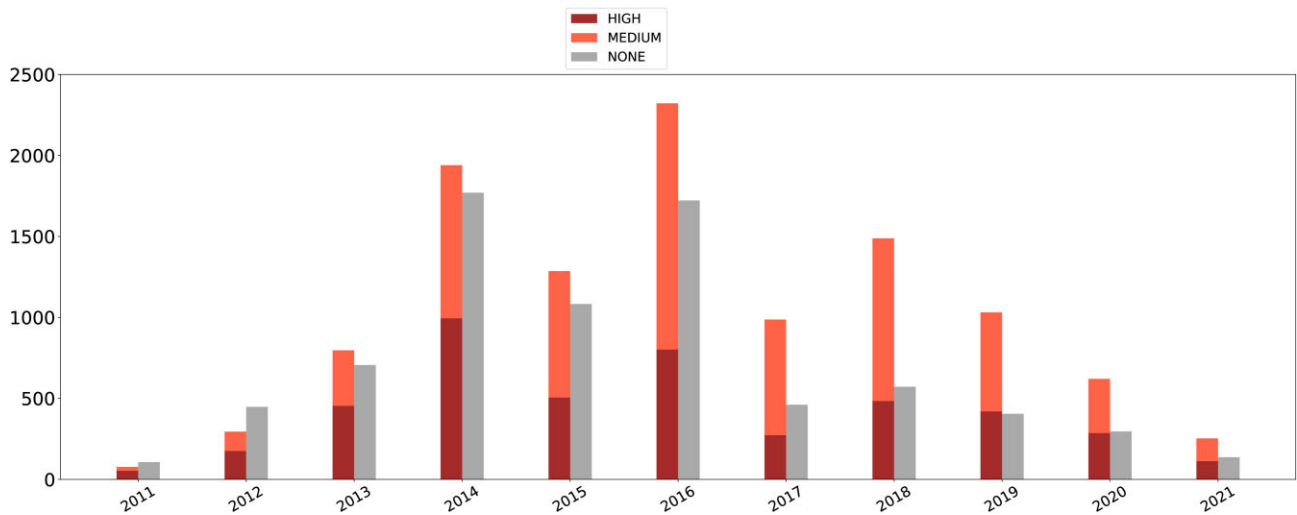   _Detection/blob/main/a-risk-estimation-study/dataset_app_hash.txt

Figure 3. This histogram shows the risk level per year on the analyzed apps. Each year has 2 bars: red/left-bottom bar for HIGH risk, orange/left-upper bar for MEDIUM risk, and gray/right bar for NONE risk.
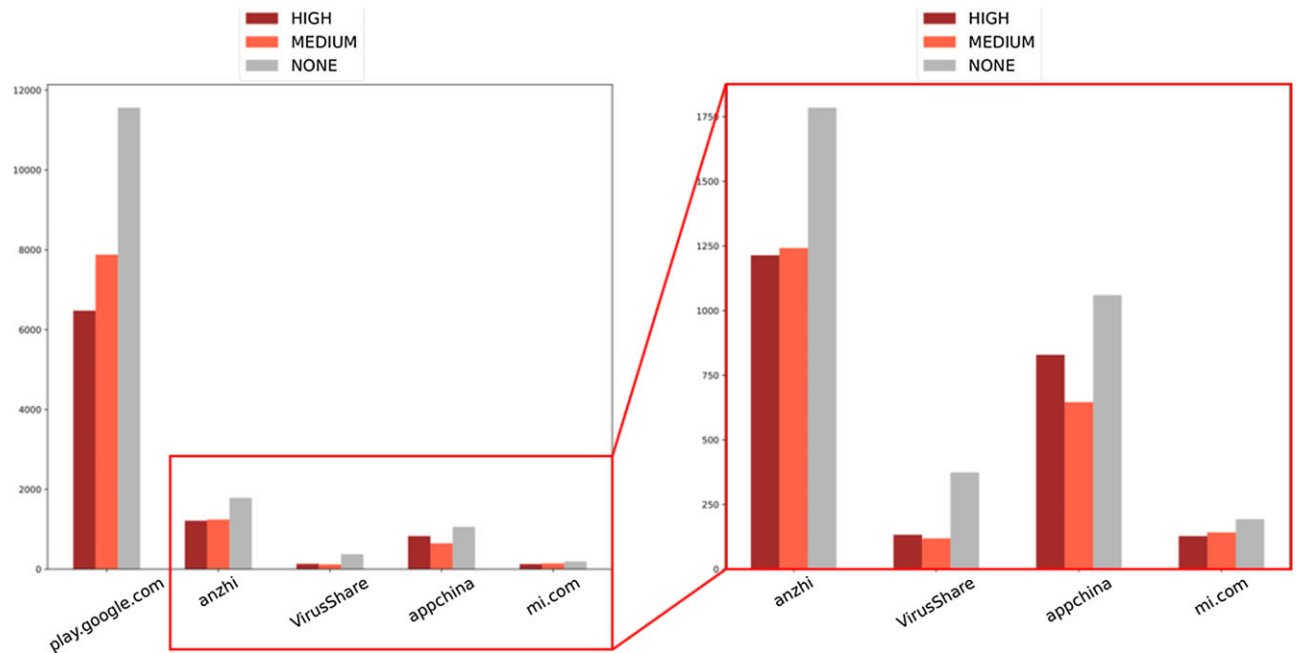

Figure 4. The histograms show the vulnerability risk levels (HIGH: red/left bar; MEDIUM: orange/central bar; NONE: gray/right bar) of the apps by markets. On the left, we plotted the main markets. On the right, we can see a detail of all markets except the Google Play Store.

Regarding applications with native code, by considering only the products in the library-identification dataset as defined in Table 4, we could determine the risk for 55% of them. Hence, about 24 000 APKs belong to at least one of our 15 products with a risk level *HIGH–MEDIUM* as reported in Fig. 2.

We also computed some statistics about the apps' year of release and belonging market.

Figure 3 shows that the apps released from 2011 to 2021 have a medium risk. Note that the *NONE* label does not mean that applications are not vulnerable at all, but that they belong to other products (i.e. no product among the selected ones is found). Expanding the analyzed products dataset may increase the number of vulnerable apps.

The market with the most vulnerable applications is Google Play, as depicted in Fig. 4. That is a reasonable result because most apps in the dataset are retrieved from the Google Play Store (Fig. 5). The reason is that Google Play only limits its checks in understanding whether an uploaded application can be classified as malware without considering vulnerabilities.

Comparison with Librarian results
To demonstrate our methodology's effectiveness, we compared our results with the ones obtained by Almanee et al. [2] by downloading their public dataset of 32 APKs [42] built in 2021. We could
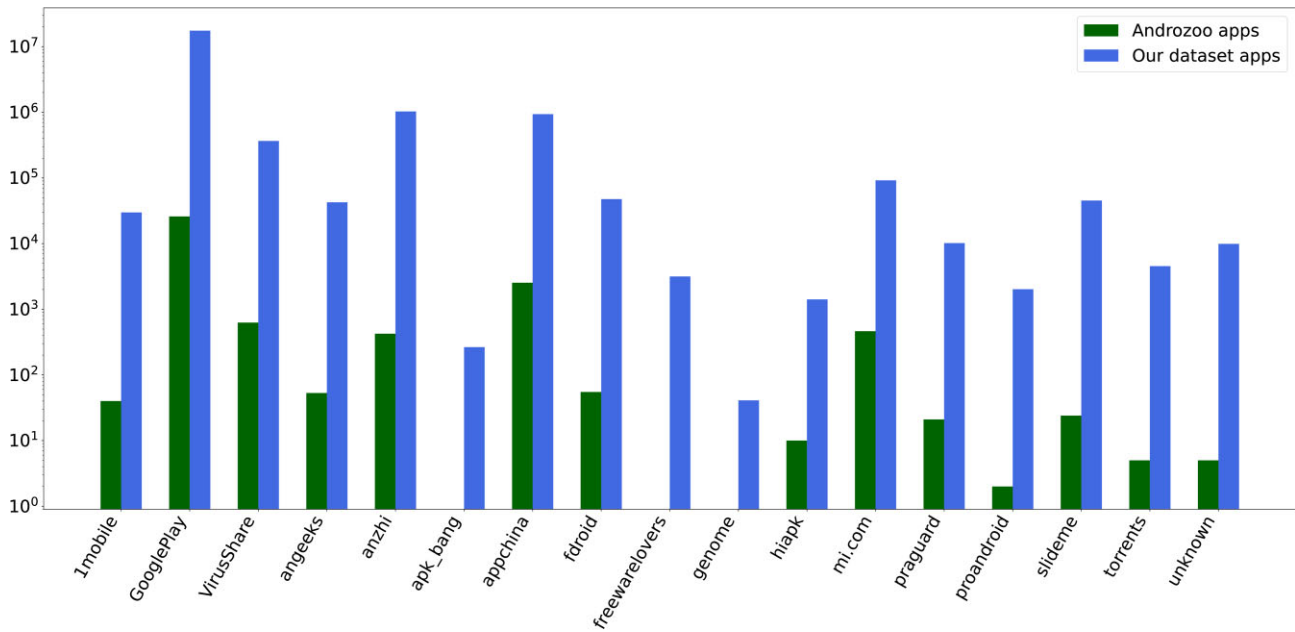
Figure 5. The histogram compares the number of apps for each market in the Androzoo dataset (green/left bar) and the apps used for vulnerability detection in our dataset (blue/right bar).
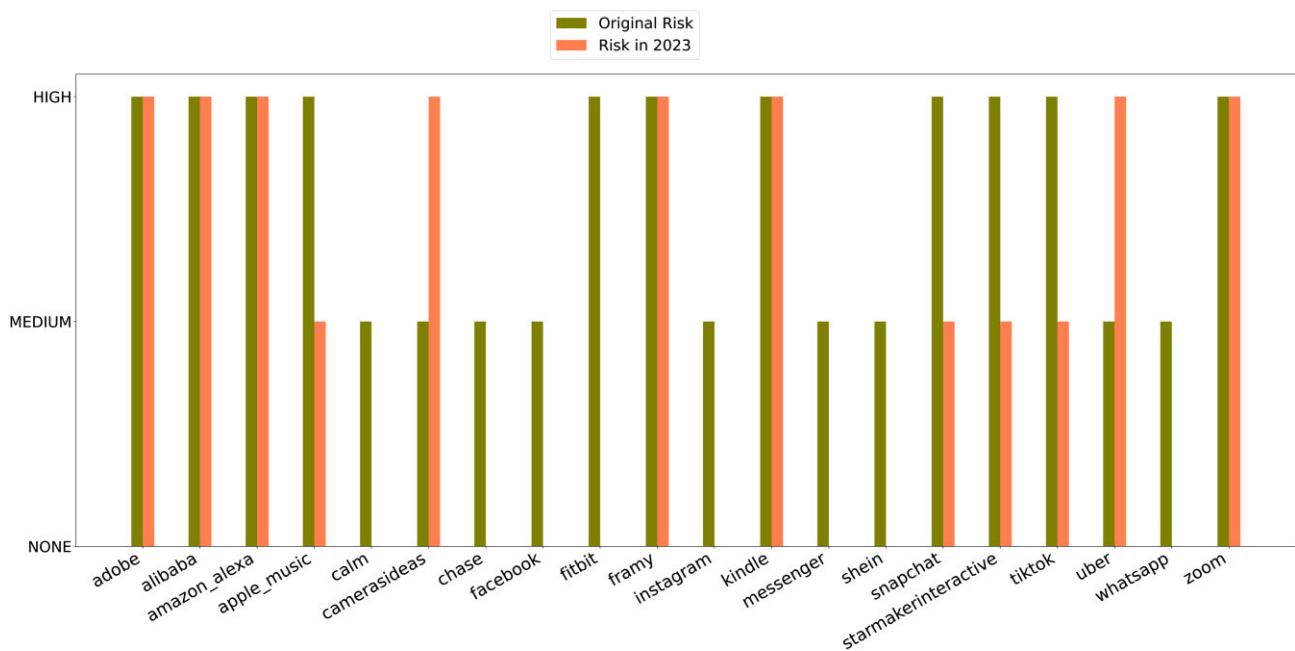

Figure 6. The histogram shows the risk level for the librarian apps (left/olive green bar) and the version on February 2023 (right/coral bar).

infer almost the same library identification results by using the same products for library identification.

While the methodology proposed by Almanee et al. was limited in saying whether the analyzed app was vulnerable (so only a result yes/no), our approach gives one of the presented risk levels in section "Risk Assessment Algorithm," we have to find a way to compare the results of the two methodologies. Hence, when our methodology gives a level to an application to which the other approach says is vulnerable, we gave the Librarian results in the same risk level set by our algorithm. On the other hand, when our methodology gives a level to an application, but the approach of Amanee et al. says it is not vulnerable, we gave the value of 0 to the Librarian results. First of all, we infer the same results as the Librarian did for all applications, except for one, which we detected as MEDIUM risk, but the Librarian says it has no vulnerabilities. From the results, we can affirm that 47% of the apps have a HIGH risk level, 41% MEDIUM risk level, and 12% of the apps do not have vulnerable native code libraries (i.e. their products do not belong to our or Librarian dataset of products; the products do not contain CVEs).

Comparison with updated Librarian dataset

The last experiment we performed was on the last version of the Librarian apps, downloaded on February 2023, used as a dataset to check whether the risk changed. As Fig. 6 shows, we can state that for the 55% of the apps, the risk was reduced; for the 10% of them, the risk remained constant, and for the 2% of them, it increased. The null results are caused by native code that does not belong to any of the selected 15 products.

As seen in Fig. 2, the selected 15 products are still insufficient to scan all apps, even though they are the most popular. Moreover, Facebook, Instagram, Messenger, and WhatsApp use native code developed by Meta instead of importing third parties as in the Librarian's version.

From the graph in Fig. 3, we have seen that the risk reduces over time, which is expected whether we consider that various vulnerabilities have been addressed over the years. However, it is also interesting that *the risk for various apps has not changed over time*, questioning the quality of vulnerability addressing in popular applications.

Summary, limitations, and future works

In this work, we proposed a simple and fast approach for vulnerability detection in Android native code by developing the first database of CVEs for vulnerability detection by easily accessing the vulnerable version of each app and its vulnerable functions. We combined this with developing a risk assessment algorithm for vulnerability management.

We demonstrated that even a simple approach like ours is efficient for vulnerable library identification, as we have been able to reproduce the same results as previous works by highlighting vulnerable applications on a much larger scale. Our methodology can aid developers and security researchers mitigate immediate risks by recommending fine-grained application patching, thus allowing them to release more secure Android applications in the different Android markets.

However, our methodology does not consider issues such as *reachability* to determine whether vulnerable functions are accessible in apps and *stripped binaries* to assess the presence of the vulnerable function's name in the binary. In fact, our solution gives security researchers and developers a risk score so they can manually check the vulnerability. To better score the functions' reachability, security researchers can refer to other tools such as DroidReach [3]. Additionally, we do not check whether or not the vulnerable library has been patched. Indeed, even though the analyzed ELF file matches a vulnerable version or the function name, developers may have patched the function's body, for which binary similarities techniques must be used. Developers can also rename the functions or obfuscate their name (as well as the content). For these cases, our whitelist approach is insufficient to determine the risk and match whether the found function is in the vulnerability database. All these issues can be addressed in future research works.

In the future, we plan to address these issues by extending the product dataset and including as many libraries as possible to check how the risk changes. Concerning library identification, we plan to extract unique functions from each version of the products and use them as features for nachine learning algorithms.

Acknowledgements

Author contribution

Silvia Lucia Sanna (Conceptualization, Data curation, Formal analysis, Methodology, Software, Writing – original draft, Writing – review & editing), Diego Soi (Writing – original draft, Writing – review & editing), Davide Maiorca (Methodology, Project administration, Writing – review & editing), Giorgio Fumera (Methodology, Writing – review & editing), Giorgio Giacinto (Methodology, Writing – review & editing).

Conflict of interest statement: The authors declare no competing interest.

Funding

References

1. StatCounter-GlobalStats. Mobile operating system Market Share Worldwide. https://gs.statcounter.com/os-market-share#monthly-200901-202309 (Last Access: 15/09/2023).

2. Almanee S, Ünal A, Payer M. *et al.* 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)*. 2021, 1347–59.

3. Borzacchiello L, Coppa E, Maiorca D. *et al.* Reach me if you can: on native vulnerability reachability in Android apps. In: V Atluri, R Di Pietro, CD Jensen, W Meng (eds.), *Computer Security—ESORICS 2022*. Cham: Springer Nature Switzerland, 2022, 701–22.

4. Software Bill of Materials. https://www.cisa.gov/sbom (Last Access: 20/03/2024).

5. Secure Software Development Framework. https://csrc.nist.gov/projects/ssdf#ssdf-practices (Last Access: 20/03/2024).

6. Android Operating System. https://source.android.com/ (Last Access: 15/05/2023).

7. Android Platform Architecture. https://developer.android.com/guide/platform (Last Access: 15/05/2023).

8. Android Platform Architecture. https://developer.android.com/guide/topics/manifest/manifest-intro (Last Access: 15/05/2023).

9. Android NDK. https://developer.android.com/ndk/guides (Last Access: 15/05/2023).

10. Common Vulnerability and Exposure (CVE). https://www.cve.org/About/Overview (Last Access: 15/05/2023).

11. Common Vulnerability Scoring System (CVSS). https://www.first.org/cvss/v3.1/user-guide (Last Access: 15/05/2023).

12. Mitre CVE. https://cve.mitre.org/ (Last Access: 15/05/2023).

13. CVE Details. https://www.cvedetails.com/ (Last Access: 15/05/2023).

14. National Vulnerability Database from NIST. https://nvd.nist.gov/ (Last Access: 15/05/2023).

15. ISO/IEC 27005:2008 Information technology — Security techniques — Information security risk management. https://www.iso.org/standard/42107.html (Last Access: 15/05/2023).

16. Allodi L, Massacci F. Comparing vulnerability severity and exploits using case-control studies. *ACM Trans Inf Syst Secur* 2014; **17**: 1–20 https://doi.org/10.1145/2630069.

17. Alves H, dos Santos Neto BF, Antunes N. Software metrics and security vulnerabilities: dataset and exploratory study. *2016 12th European Dependable Computing Conference (EDCC)*.2016, 37–44. https://doi.org/10.1109/EDCC.2016.34.

18. Medeiros N, Ivaki NR, Costa P. *et al.* Software metrics as indicators of security vulnerabilities. *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. 2017, 216–27.

19. Xiao Y, Chen B, Yu C. *et al.* MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures. In: *29th USENIX Security*

Symposium (USENIX Security 20). Boston: USENIX Association, 2020, 1165–82.

20. Du X, Chen B, Li Y. *et al.* LEOPARD: identifying vulnerable code for vulnerability assessment through program metrics. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, Montréal: IEEE/ACM, 60–71.

21. Ruggia A, Possemato A, Dambra S. *et al.* The dark side of native code on android. *TechRxiv* 2022, **1**: 1–33.

22. Fedler R, Kulicke M, Schütte J. Native code execution control for attack mitigation on android. In: *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. SPSM '13*. New York, NY: Association for Computing Machinery, 2013, 15–20.

23. Mobile security framework (MobSF). https://github.com/MobSF/Mobile-Security-Framework-MobSF (Last Access: 27/07/2023).

24. Quick Android Review Kit (Qark). https://github.com/linkedin/qark (Last Access: 27/07/2023).

25. Pagano F, Romdhana A, Caputo D. *et al.* SEBASTiAn: a static and extensible black-box application security testing tool for iOS and Android applications. *SoftwareX* 2023; **23**: 101448. https://doi.org/10.1016/j.softx.2023.101448.

26. Android greybox fuzzing with AFL++ Frida mode. https://blog.quarkslab.com/android-greybox-fuzzing-with-afl-frida-mode.html (Last Access: 27/07/2023)

27. android-afl. https://github.com/ele7enxxh/android-afl/blob/master/README.md (Last Access: 27/07/2023).

28. Fuzzing with libfuzzer. https://source.android.com/docs/security/test/libfuzzer (Last Access: 27/07/2023).

29. Liu B, Zhang C, Gong G. *et al.* FANS: fuzzing android native system services via automated interface analysis. In: *29th USENIX Security Symposium (USENIX Security 20)*. Berkeley, California: USENIX Association, 2020, 307–23.

30. Brant CD, Yavuz T. A Study on the Testing of Android Security Patches. In: *2022 IEEE Conference on Communications and Network Security (CNS)*. Austin, TX: IEEE, 2022, 217–25.

31. Ma Z, Wang H, Guo Y. *et al.* LibRadar: fast and accurate detection of third-party libraries in Android apps. In: *2016 IEEE/ACM 38th Interna-tional Conference on Software Engineering Companion (ICSE-C)*. Austin, TX: IEEE/ACM, 2016, 653–6.

32. Li M, Wang W, Wang P. *et al.* LibD: scalable and precise third-party library detection in Android markets. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. Buenos Aires, Argentina: IEEE/ACM, 2017, 335–46.

33. Afonso VM, de Geus PL, Bianchi A. *et al.* Going native: using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In: *Network and Distributed System Security Symposium*. Reston, VA: Internet Society. 2016.

34. Sun M, Tan G. *NativeGuard: protecting android applications from third-party native libraries. WiSec '14*, New York, NY: Association for Computing Machinery, 2014, 165–76.

35. Zhou Y, Patel K, Wu L. *et al.* Hybrid user-level sandboxing of third-party Android apps. ASIA CCS '15. New York, NY: Association for Computing Machinery, 2015, 19–30.

36. Xue L, Qian C, Zhou H. *et al.* NDroid: toward tracking information flows across multiple Android contexts. *IEEE T Inf Foren Sec* 2019; **14**: 814–28. https://doi.org/10.1109/TIFS.2018.2866347.

37. Alam S, Qu Z, Riley RD. *et al.* DroidNative: semantic-based detection of Android native code malware. *CoRR* 2016. **abs/1602.04693** 18

38. Narayanan A, Chen L, Chan CK. AdDetect: automated detection of android ad libraries using semantic analysis. In: *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. 2014, 1–6. Singapore: IEEE Xplore.

39. Zahan N, Lin E, Tamanna M. *et al.* Software bills of materials are required. are we there yet?. *IEEE Secur Priv* 2023; **21**: 82–8.

40. NIST. EXECUTIVE ORDER 14028, IMPROVING THE NATION'S CYBERSECURITY. https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity (Last Access: 20/03/2024).

41. Pwntools. http://docs.pwntools.com/en/latest/ (Last Access: 10/07/2023).

42. Librarian GitHub Repository. https://github.com/salmanee/Librarian (Last Access: 08/09/2023).